Ironic Documentation

Release 20.1.3.dev2

OpenStack Foundation

Oct 26, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

Ironic is an OpenStack project which provisions bare metal (as opposed to virtual) machines. It may be used independently or as part of an OpenStack Cloud, and integrates with the OpenStack Identity (keystone), Compute (nova), Network (neutron), Image (glance), and Object (swift) services.

The Bare Metal service manages hardware through both common (eg. PXE and IPMI) and vendor-specific remote management protocols. It provides the cloud operator with a unified interface to a heterogeneous fleet of servers while also providing the Compute service with an interface that allows physical servers to be managed as though they were virtual machines.

This documentation is continually updated and may not represent the state of the project at any specific prior release. To access documentation for a previous release of ironic, append the OpenStack release name to the URL; for example, the `ocata` release is available at https://docs.openstack.org/ironic/ocata/.

Found a bug in one of our projects? Please see *Bug Reporting and Triaging Guide*.

Would like to engage with the community? See *Bare Metal Community*.

CHAPTER

TWO

INSTALLATION GUIDE

2.1 Bare Metal Service Installation Guide

The Bare Metal service is a collection of components that provides support to manage and provision physical machines.

This chapter assumes a working setup of OpenStack following the OpenStack Installation Guides. It contains the following sections:

2.1.1 Bare Metal service overview

The Bare Metal service, codenamed `ironic`, is a collection of components that provides support to manage and provision physical machines.

Bare Metal service components

The Bare Metal service includes the following components:

ironic-api A RESTful API that processes application requests by sending them to the ironic-conductor over remote procedure call (RPC). Can be run through WSGI or as a separate process.

ironic-conductor Adds/edits/deletes nodes; powers on/off nodes with IPMI or other vendor-specific protocol; provisions/deploys/cleans bare metal nodes.

ironic-conductor uses *drivers* to execute operations on hardware.

ironic-python-agent A python service which is run in a temporary ramdisk to provide ironic-conductor and ironic-inspector services with remote access, in-band hardware control, and hardware introspection.

Additionally, the Bare Metal service has certain external dependencies, which are very similar to other OpenStack services:

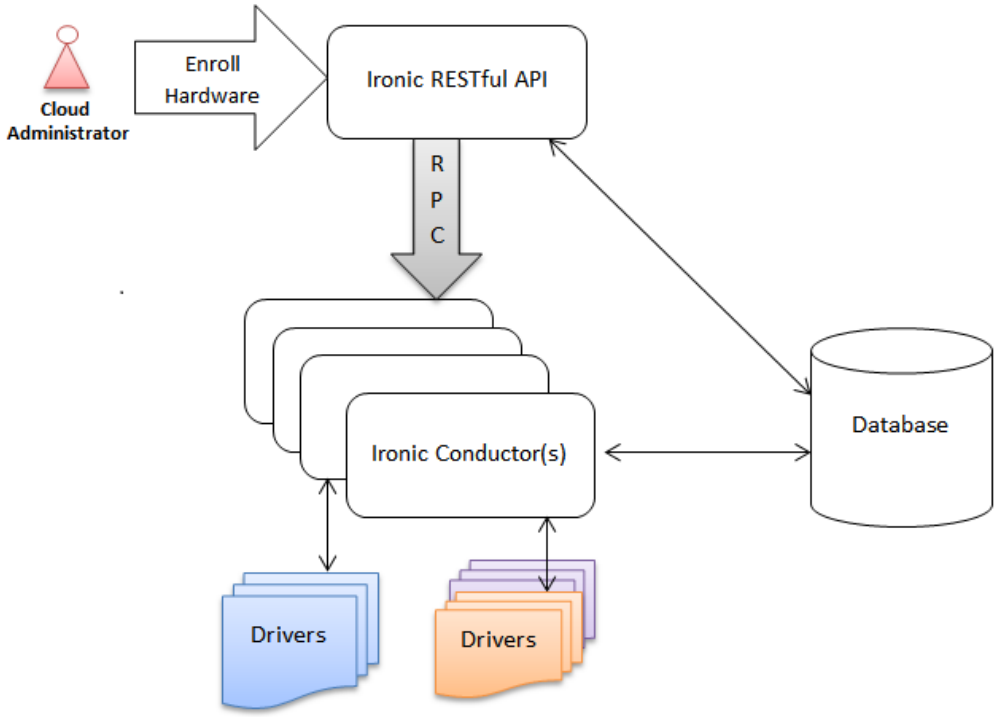
- A database to store hardware information and state. You can set the database back-end type and location. A simple approach is to use the same database back end as the Compute service. Another approach is to use a separate database back-end to further isolate bare metal resources (and associated metadata) from users.
- An oslo.messaging compatible queue, such as RabbitMQ. It may use the same implementation as that of the Compute service, but that is not a requirement. Used to implement RPC between ironic-api and ironic-conductor.

Deployment architecture

The Bare Metal RESTful API service is used to enroll hardware that the Bare Metal service will manage. A cloud administrator usually registers it, specifying their attributes such as MAC addresses and IPMI credentials. There can be multiple instances of the API service.

The *ironic-conductor* process does the bulk of the work. For security reasons, it is advisable to place it on an isolated host, since it is the only service that requires access to both the data plane and IPMI control plane.

There can be multiple instances of the conductor service to support various class of drivers and also to manage fail over. Instances of the conductor service should be on separate nodes. Each conductor can itself run many drivers to operate heterogeneous hardware. This is depicted in the following figure.



The API exposes a list of supported drivers and the names of conductor hosts servicing them.

Interaction with OpenStack components

The Bare Metal service may, depending upon configuration, interact with several other OpenStack services. This includes:

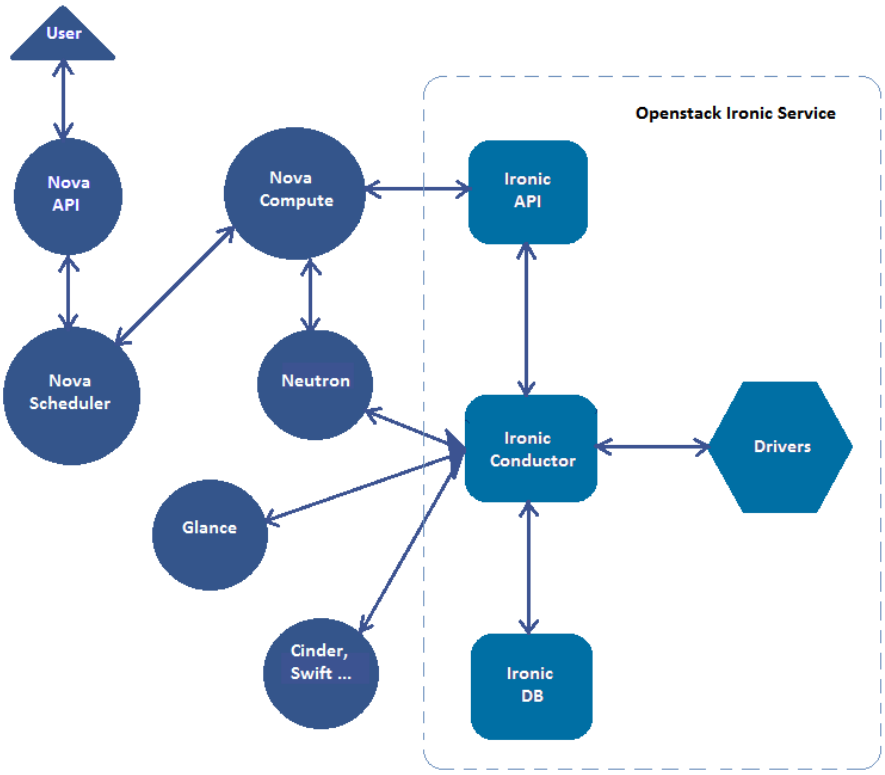
- the OpenStack Telemetry module (`ceilometer`) for consuming the IPMI metrics
- the OpenStack Identity service (`keystone`) for request authentication and to locate other OpenStack services
- the OpenStack Image service (`glance`) from which to retrieve images and image meta-data
- the OpenStack Networking service (`neutron`) for DHCP and network configuration

- the OpenStack Compute service (`nova`) works with the Bare Metal service and acts as a user-facing API for instance management, while the Bare Metal service provides the admin/operator API for hardware management. The OpenStack Compute service also provides scheduling facilities (matching flavors <-> images <-> hardware), tenant quotas, IP assignment, and other services which the Bare Metal service does not, in and of itself, provide.
- the OpenStack Object Storage (`swift`) provides temporary storage for the configdrive, user images, deployment logs and inspection data.

Logical architecture

The diagram below shows the logical architecture. It shows the basic components that form the Bare Metal service, the relation of the Bare Metal service with other OpenStack services and the logical flow of a boot instance request resulting in the provisioning of a physical server.

Figure 1.2. Logical Architecture



A users request to boot an instance is passed to the Compute service via the Compute API and the Compute Scheduler. The Compute service uses the *ironic virt driver* to hand over this request to the Bare Metal service, where the request passes from the Bare Metal API, to the Conductor, to a Driver to successfully provision a physical server for the user.

Just as the Compute service talks to various OpenStack services like Image, Network, Object Store etc to provision a virtual machine instance, here the Bare Metal service talks to the same OpenStack services for image, network and other resource needs to provision a bare metal instance.

See *Understanding Bare Metal Deployment* for a more detailed breakdown of a typical deployment process.

Associated projects

Optionally, one may wish to utilize the following associated projects for additional functionality:

python-ironicclient A command-line interface (CLI) and python bindings for interacting with the Bare Metal service.

ironic-ui Horizon dashboard, providing graphical interface (GUI) for the Bare Metal API.

ironic-inspector An associated service which performs in-band hardware introspection by PXE booting unregistered hardware into the ironic-python-agent ramdisk.

diskimage-builder A related project to help facilitate the creation of ramdisks and machine images, such as those running the ironic-python-agent.

bifrost A set of Ansible playbooks that automates the task of deploying a base image onto a set of known hardware using ironic in a standalone mode.

2.1.2 Reference Deploy Architectures

This section covers the way we recommend the Bare Metal service to be deployed and managed. It is assumed that a reader has already gone through *Bare Metal Service User Guide*. It may be also useful to try *Deploying Ironic with DevStack* first to get better familiar with the concepts used in this guide.

Common Considerations

This section covers considerations that are equally important to all described architectures.

Components

As explained in *Bare Metal service overview*, the Bare Metal service has three components.

- The Bare Metal API service (`ironic-api`) should be deployed in a similar way as the control plane API services. The exact location will depend on the architecture used.
- The Bare Metal conductor service (`ironic-conductor`) is where most of the provisioning logic lives. The following considerations are the most important when deciding on the way to deploy it:
  - The conductor manages a certain proportion of nodes, distributed to it via a hash ring. This includes constantly polling these nodes for their current power state and hardware sensor data (if enabled and supported by hardware, see *Collecting sensor data* for an example).
  - The conductor needs access to the management controller of each node it manages.
  - The conductor co-exists with TFTP (for PXE) and/or HTTP (for iPXE) services that provide the kernel and ramdisk to boot the nodes. The conductor manages them by writing files to their root directories.
  - If serial console is used, the conductor launches console processes locally. If the `nova-serialproxy` service (part of the Compute service) is used, it has to be able to reach the conductors. Otherwise, they have to be directly accessible by the users.
  - There must be mutual connectivity between the conductor and the nodes being deployed or cleaned. See *Networking* for details.
- The provisioning ramdisk which runs the `ironic-python-agent` service on start up.

> **Warning:** The `ironic-python-agent` service is not intended to be used or executed anywhere other than a provisioning/cleaning/rescue ramdisk.

Hardware and drivers

The Bare Metal service strives to provide the best support possible for a variety of hardware. However, not all hardware is supported equally well. It depends on both the capabilities of hardware itself and the available drivers. This section covers various considerations related to the hardware interfaces. See *Enabling drivers and hardware types* for a detailed introduction into hardware types and interfaces before proceeding.

Power and management interfaces

The minimum set of capabilities that the hardware has to provide and the driver has to support is as follows:

1. getting and setting the power state of the machine
2. getting and setting the current boot device
3. booting an image provided by the Bare Metal service (in the simplest case, support booting using PXE and/or iPXE)

Note: Strictly speaking, it is possible to make the Bare Metal service provision nodes without some of these capabilities via some manual steps. It is not the recommended way of deployment, and thus it is not covered in this guide.

Once you make sure that the hardware supports these capabilities, you need to find a suitable driver. Most of enterprise-grade hardware has support for IPMI and thus can utilize *IPMI driver*. Some newer hardware also supports *Redfish driver*. Several vendors provide more specific drivers that usually provide additional capabilities. Check *Drivers, Hardware Types and Hardware Interfaces* to find the most suitable one.

Boot interface

The boot interface of a node manages booting of both the deploy ramdisk and the user instances on the bare metal node. The deploy interface orchestrates the deployment and defines how the image gets transferred to the target disk.

The main alternatives are to use PXE/iPXE or virtual media - see *Boot interfaces* for a detailed explanation. If a virtual media implementation is available for the hardware, it is recommended using it for better scalability and security. Otherwise, it is recommended to use iPXE, when it is supported by target hardware.

Hardware specifications

The Bare Metal services does not impose too many restrictions on the characteristics of hardware itself. However, keep in mind that

- By default, the Bare Metal service will pick the smallest hard drive that is larger than 4 GiB for deployment. Another hard drive can be used, but it requires setting *root device hints*.

  Note: This device does not have to match the boot device set in BIOS (or similar firmware).

- The machines should have enough RAM to fit the deployment/cleaning ramdisk to run. The minimum varies greatly depending on the way the ramdisk was built. For example, *tinyipa*, the TinyCoreLinux-based ramdisk used in the CI, only needs 400 MiB of RAM, while ramdisks built by *diskimage-builder* may require 3 GiB or more.

Image types

The Bare Metal service can deploy two types of images:

- *Whole-disk* images that contain a complete partitioning table with all necessary partitions and a bootloader. Such images are the most universal, but may be harder to build.
- *Partition images* that contain only the root partition. The Bare Metal service will create the necessary partitions and install a boot loader, if needed.

Warning: Partition images are only supported with GNU/Linux operating systems.

Warning: If you plan on using local boot, your partition images must contain GRUB2 bootloader tools to enable ironic to set up the bootloader during deploy.

Local vs network boot

The Bare Metal service supports booting user instances either using a local bootloader or using the drivers boot interface (e.g. via PXE or iPXE protocol in case of the `pxe` interface).

Network boot cannot be used with certain architectures (for example, when no tenant networks have access to the control plane).

Additional considerations are related to the `pxe` boot interface, and other boot interfaces based on it:

- Local boot makes nodes boot process independent of the Bare Metal conductor managing it. Thus, nodes are able to reboot correctly, even if the Bare Metal TFTP or HTTP service is down.
- Network boot (and iPXE) must be used when booting nodes from remote volumes, if the driver does not support attaching volumes out-of-band.

The default boot option for the cloud can be changed via the Bare Metal service configuration file, for example:

```
[deploy]
default_boot_option = local
```

This default can be overridden by setting the `boot_option` capability on a node. See *Local boot with partition images* for details.

Note: Currently, local boot is used by default. Its safer to set the `default_boot_option` explicitly.

Networking

There are several recommended network topologies to be used with the Bare Metal service. They are explained in depth in specific architecture documentation. However, several considerations are common for all of them:

- There has to be a *provisioning* network, which is used by nodes during the deployment process. If allowed by the architecture, this network should not be accessible by end users, and should not have access to the internet.
- There has to be a *cleaning* network, which is used by nodes during the cleaning process.
- There should be a *rescuing* network, which is used by nodes during the rescue process. It can be skipped if the rescue process is not supported.

Note: In the majority of cases, the same network should be used for cleaning, provisioning and rescue for simplicity.

Unless noted otherwise, everything in these sections apply to all three networks.

- The baremetal nodes must have access to the Bare Metal API while connected to the provisioning/cleaning/rescuing network.

  Note: Only two endpoints need to be exposed there:

  ```
  GET /v1/lookup
  POST /v1/heartbeat/[a-z0-9\-]+
  ```

  You may want to limit access from this network to only these endpoints, and make these endpoint not accessible from other networks.

- If the `pxe` boot interface (or any boot interface based on it) is used, then the baremetal nodes should have untagged (access mode) connectivity to the provisioning/cleaning/rescuing networks. It allows PXE firmware, which does not support VLANs, to communicate with the services required for provisioning.

  Note: It depends on the *network interface* whether the Bare Metal service will handle it automatically. Check the networking documentation for the specific architecture.

  Sometimes it may be necessary to disable the spanning tree protocol delay on the switch - see *DHCP during PXE or iPXE is inconsistent or unreliable*.

- The Baremetal nodes need to have access to any services required for provisioning/cleaning/rescue, while connected to the provisioning/cleaning/rescuing network. This may include:
  - a TFTP server for PXE boot and also an HTTP server when iPXE is enabled
  - either an HTTP server or the Object Storage service in case of the `direct` deploy interface and some virtual media boot interfaces
- The Baremetal Conductors need to have access to the booted baremetal nodes during provisioning/cleaning/rescue. A conductor communicates with an internal API, provided by **ironic-python-agent**, to conduct actions on nodes.

HA and Scalability

ironic-api

The Bare Metal API service is stateless, and thus can be easily scaled horizontally. It is recommended to deploy it as a WSGI application behind e.g. Apache or another WSGI container.

Note: This service accesses the ironic database for reading entities (e.g. in response to `GET /v1/nodes` request) and in rare cases for writing.

ironic-conductor

High availability

The Bare Metal conductor service utilizes the active/active HA model. Every conductor manages a certain subset of nodes. The nodes are organized in a hash ring that tries to keep the load spread more or less uniformly across the conductors. When a conductor is considered offline, its nodes are taken over by other conductors. As a result of this, you need at least 2 conductor hosts for an HA deployment.

Performance

Conductors can be resource intensive, so it is recommended (but not required) to keep all conductors separate from other services in the cloud. The minimum required number of conductors in a deployment depends on several factors:

- the performance of the hardware where the conductors will be running,
- the speed and reliability of the management controller of the bare metal nodes (for example, handling slower controllers may require having less nodes per conductor),
- the frequency, at which the management controllers are polled by the Bare Metal service (see the `sync_power_state_interval` option),
- the bare metal driver used for nodes (see *Hardware and drivers* above),
- the network performance,
- the maximum number of bare metal nodes that are provisioned simultaneously (see the `max_concurrent_builds` option for the Compute service).

We recommend a target of **100** bare metal nodes per conductor for maximum reliability and performance. There is some tolerance for a larger number per conductor. However, it was reported[1][2] that reliability degrades when handling approximately 300 bare metal nodes per conductor.

Disk space

Each conductor needs enough free disk space to cache images it uses. Depending on the combination of the deploy interface and the boot option, the space requirements are different:

- The deployment kernel and ramdisk are always cached during the deployment.
- When `[agent]image_download_source` is set to `http` and Glance is used, the conductor will download instances images locally to serve them from its HTTP server. Use `swift` to publish images using temporary URLs and convert them on the nodes side.

  When `[agent]image_download_source` is set to `local`, it will happen even for HTTP(s) URLs. For standalone case use `http` to avoid unnecessary caching of images.

  In both cases a cached image is converted to raw if `force_raw_images` is `True` (the default).

[1] http://lists.openstack.org/pipermail/openstack-dev/2017-June/118033.html
[2] http://lists.openstack.org/pipermail/openstack-dev/2017-June/118327.html

> **Note:** `image_download_source` can also be provided in the nodes `driver_info` or `instance_info`. See *Deploy with custom HTTP servers*.

- When network boot is used, the instance image kernel and ramdisk are cached locally while the instance is active.

Note: All images may be stored for some time after they are no longer needed. This is done to speed up simultaneous deployments of many similar images. The caching can be configured via the `image_cache_size` and `image_cache_ttl` configuration options in the `pxe` group.

Other services

When integrating with other OpenStack services, more considerations may need to be applied. This is covered in other parts of this guide.

Scenarios

Small cloud with trusted tenants

Story

As an operator I would like to build a small cloud with both virtual and bare metal instances or add bare metal provisioning to my existing small or medium scale single-site OpenStack cloud. The expected number of bare metal machines is less than 100, and the rate of provisioning and unprovisioning is expected to be low. All users of my cloud are trusted by me to not conduct malicious actions towards each other or the cloud infrastructure itself.

As a user I would like to occasionally provision bare metal instances through the Compute API by selecting an appropriate Compute flavor. I would like to be able to boot them from images provided by the Image service or from volumes provided by the Volume service.

Components

This architecture assumes an OpenStack installation with the following components participating in the bare metal provisioning:

- The Compute service manages bare metal instances.
- The Networking service provides DHCP for bare metal instances.
- The Image service provides images for bare metal instances.

The following services can be optionally used by the Bare Metal service:

- The Volume service provides volumes to boot bare metal instances from.
- The Bare Metal Introspection service simplifies enrolling new bare metal machines by conducting in-band introspection.

Node roles

An OpenStack installation in this guide has at least these three types of nodes:

- A *controller* node hosts the control plane services.
- A *compute* node runs the virtual machines and hosts a subset of Compute and Networking components.
- A *block storage* node provides persistent storage space for both virtual and bare metal nodes.

The *compute* and *block storage* nodes are configured as described in the installation guides of the Compute service and the Volume service respectively. The *controller* nodes host the Bare Metal service components.

Networking

The networking architecture will highly depend on the exact operating requirements. This guide expects the following existing networks: *control plane*, *storage* and *public*. Additionally, two more networks will be needed specifically for bare metal provisioning: *bare metal* and *management*.

Control plane network

The *control plane network* is the network where OpenStack control plane services provide their public API.

The Bare Metal API will be served to the operators and to the Compute service through this network.

Public network

The *public network* is used in a typical OpenStack deployment to create floating IPs for outside access to instances. Its role is the same for a bare metal deployment.

Note: Since, as explained below, bare metal nodes will be put on a flat provider network, it is also possible to organize direct access to them, without using floating IPs and bypassing the Networking service completely.

Bare metal network

The *Bare metal network* is a dedicated network for bare metal nodes managed by the Bare Metal service.

This architecture uses *flat bare metal networking*, in which both tenant traffic and technical traffic related to the Bare Metal service operation flow through this one network. Specifically, this network will serve as the *provisioning*, *cleaning* and *rescuing* network. It will also be used for introspection via the Bare Metal Introspection service. See *common networking considerations* for an in-depth explanation of the networks used by the Bare Metal service.

DHCP and boot parameters will be provided on this network by the Networking services DHCP agents.

For booting from volumes this network has to have a route to the *storage network*.

Management network

Management network is an independent network on which BMCs of the bare metal nodes are located.

The `ironic-conductor` process needs access to this network. The tenants of the bare metal nodes must not have access to it.

Note: The *direct deploy interface* and certain *Drivers, Hardware Types and Hardware Interfaces* require the *management network* to have access to the Object storage service backend.

Controllers

A *controller* hosts the OpenStack control plane services as described in the control plane design guide. While this architecture allows using *controllers* in a non-HA configuration, it is recommended to have at least three of them for HA. See *HA and Scalability* for more details.

Bare Metal services

The following components of the Bare Metal service are installed on a *controller* (see *components of the Bare Metal service*):

- The Bare Metal API service either as a WSGI application or the `ironic-api` process. Typically, a load balancer, such as HAProxy, spreads the load between the API instances on the *controllers*.

  The API has to be served on the *control plane network*. Additionally, it has to be exposed to the *bare metal network* for the ramdisk callback API.

- The `ironic-conductor` process. These processes work in active/active HA mode as explained in *HA and Scalability*, thus they can be installed on all *controllers*. Each will handle a subset of bare metal nodes.

  The `ironic-conductor` processes have to have access to the following networks:

  - *control plane* for interacting with other services
  - *management* for contacting nodes BMCs
  - *bare metal* for contacting deployment, cleaning or rescue ramdisks

- TFTP and HTTP service for booting the nodes. Each `ironic-conductor` process has to have a matching TFTP and HTTP service. They should be exposed only to the *bare metal network* and must not be behind a load balancer.

- The `nova-compute` process (from the Compute service). These processes work in active/active HA mode when dealing with bare metal nodes, thus they can be installed on all *controllers*. Each will handle a subset of bare metal nodes.

  Note: There is no 1-1 mapping between `ironic-conductor` and `nova-compute` processes, as they communicate only through the Bare Metal API service.

- The networking-baremetal ML2 plugin should be loaded into the Networking service to assist with binding bare metal ports.

  The ironic-neutron-agent service should be started as well.

- If the Bare Metal introspection is used, its `ironic-inspector` process has to be installed on all *controllers*. Each such process works as both Bare Metal Introspection API and conductor service. A load balancer should be used to spread the API load between *controllers*.

  The API has to be served on the *control plane network*. Additionally, it has to be exposed to the *bare metal network* for the ramdisk callback API.

Shared services

A *controller* also hosts two services required for the normal operation of OpenStack:

- Database service (MySQL/MariaDB is typically used, but other enterprise-grade database solutions can be used as well).

  All Bare Metal service components need access to the database service.

- Message queue service (RabbitMQ is typically used, but other enterprise-grade message queue brokers can be used as well).

  Both Bare Metal API (WSGI application or `ironic-api` process) and the `ironic-conductor` processes need access to the message queue service. The Bare Metal Introspection service does not need it.

Note: These services are required for all OpenStack services. If youre adding the Bare Metal service to your cloud, you may reuse the existing database and messaging queue services.

Bare metal nodes

Each bare metal node must be capable of booting from network, virtual media or other boot technology supported by the Bare Metal service as explained in *Boot interface*. Each node must have one NIC on the *bare metal network*, and this NIC (and **only** it) must be configured to be able to boot from network. This is usually done in the *BIOS setup* or a similar firmware configuration utility. There is no need to alter the boot order, as it is managed by the Bare Metal service. Other NICs, if present, will not be managed by OpenStack.

The NIC on the *bare metal network* should have untagged connectivity to it, since PXE firmware usually does not support VLANs - see *Networking* for details.

Storage

If your hardware **and** its bare metal *driver* support booting from remote volumes, please check the driver documentation for information on how to enable it. It may include routing *management* and/or *bare metal* networks to the *storage network*.

In case of the standard *PXE boot*, booting from remote volumes is done via iPXE. In that case, the Volume storage backend must support iSCSI protocol, and the *bare metal network* has to have a route to the *storage network*. See *Boot From Volume* for more details.

2.1.3 Install and configure the Bare Metal service

This section describes how to install and configure the Bare Metal service, code-named ironic, manually from packages on one of the three popular families of Linux distributions.

Alternatively, you can use one of the numerous projects that install ironic. One of them is provided by the bare metal team:

- Bifrost installs ironic in the standalone mode (without the rest of OpenStack).

More installation projects are developed by other OpenStack teams:

- Kolla can install ironic in containers as part of OpenStack.
- OpenStack-Ansible has a role to install ironic.
- TripleO uses ironic for provisioning bare metal nodes and can also be used to install ironic.

Contents

Install and configure for Red Hat Enterprise Linux and CentOS

This section describes how to install and configure the Bare Metal service for Red Hat Enterprise Linux 8 and CentOS 8.

Install and configure prerequisites

The Bare Metal service is a collection of components that provides support to manage and provision physical machines. You can configure these components to run on separate nodes or the same node. In this guide, the components run on one node, typically the Compute Services compute node.

It assumes that the Identity, Image, Compute, and Networking services have already been set up.

Set up the database for Bare Metal

The Bare Metal service stores information in a database. This guide uses the MySQL database that is used by other OpenStack services.

1. In MySQL, create an `ironic` database that is accessible by the `ironic` user. Replace `IRONIC_DBPASSWORD` with a suitable password:

```
# mysql -u root -p
mysql> CREATE DATABASE ironic CHARACTER SET utf8;
mysql> GRANT ALL PRIVILEGES ON ironic.* TO 'ironic'@'localhost' \
       IDENTIFIED BY 'IRONIC_DBPASSWORD';
mysql> GRANT ALL PRIVILEGES ON ironic.* TO 'ironic'@'%' \
       IDENTIFIED BY 'IRONIC_DBPASSWORD';
```

Install and configure components

1. Install from packages (using dnf)

```
# dnf install openstack-ironic-api openstack-ironic-conductor python3-
↪ironicclient
```

2. Enable services

```
# systemctl enable openstack-ironic-api openstack-ironic-conductor
# systemctl start openstack-ironic-api openstack-ironic-conductor
```

The Bare Metal service is configured via its configuration file. This file is typically located at `/etc/ironic/ironic.conf`.

Although some configuration options are mentioned here, it is recommended that you review all the *Sample Configuration File* so that the Bare Metal service is configured for your needs.

It is possible to set up an ironic-api and an ironic-conductor services on the same host or different hosts. Users also can add new ironic-conductor hosts to deal with an increasing number of bare metal nodes. But the additional ironic-conductor services should be at the same version as that of existing ironic-conductor services.

Configuring ironic-api service

1. The Bare Metal service stores information in a database. This guide uses the MySQL database that is used by other OpenStack services.

   Configure the location of the database via the `connection` option. In the following, replace `IRONIC_DBPASSWORD` with the password of your `ironic` user, and replace `DB_IP` with the IP address where the DB server is located:

```
[database]

# The SQLAlchemy connection string used to connect to the
```

(continues on next page)

(continued from previous page)

```
# database (string value)
connection=mysql+pymysql://ironic:IRONIC_DBPASSWORD@DB_IP/ironic?
↪charset=utf8
```

2. Configure the ironic-api service to use the RabbitMQ message broker by setting the following option. Replace RPC_* with appropriate address details and credentials of RabbitMQ server:

```
[DEFAULT]

# A URL representing the messaging driver to use and its full
# configuration. (string value)
transport_url = rabbit://RPC_USER:RPC_PASSWORD@RPC_HOST:RPC_PORT/
```

Alternatively, you can use JSON RPC for interactions between ironic-conductor and ironic-api. Enable it in the configuration and provide the keystone credentials to use for authentication:

```
[DEFAULT]

rpc_transport = json-rpc

[json_rpc]

# Authentication type to load (string value)
auth_type = password

# Authentication URL (string value)
auth_url=https://IDENTITY_IP:5000/

# Username (string value)
username=ironic

# User's password (string value)
password=IRONIC_PASSWORD

# Project name to scope to (string value)
project_name=service

# Domain ID containing project (string value)
project_domain_id=default

# User's domain id (string value)
user_domain_id=default
```

If you use port other than the default 8089 for JSON RPC, you have to configure it, for example:

```
[json_rpc]
port = 9999
```

3. Configure the ironic-api service to use these credentials with the Identity service. Replace PUBLIC_IDENTITY_IP with the public IP of the Identity server, PRIVATE_IDENTITY_IP with the private IP of the Identity server and replace IRONIC_PASSWORD with the password you chose

for the `ironic` user in the Identity service:

```
[DEFAULT]

# Authentication strategy used by ironic-api: one of
# "keystone" or "noauth". "noauth" should not be used in a
# production environment because all authentication will be
# disabled. (string value)
auth_strategy=keystone

[keystone_authtoken]

# Authentication type to load (string value)
auth_type=password

# Complete public Identity API endpoint (string value)
www_authenticate_uri=http://PUBLIC_IDENTITY_IP:5000

# Complete admin Identity API endpoint. (string value)
auth_url=http://PRIVATE_IDENTITY_IP:5000

# Service username. (string value)
username=ironic

# Service account password. (string value)
password=IRONIC_PASSWORD

# Service tenant name. (string value)
project_name=service

# Domain name containing project (string value)
project_domain_name=Default

# User's domain name (string value)
user_domain_name=Default
```

4. Create the Bare Metal service database tables:

```
$ ironic-dbsync --config-file /etc/ironic/ironic.conf create_schema
```

5. Restart the ironic-api service:

   Fedora/RHEL8/CentOS8/SUSE:

```
sudo systemctl restart openstack-ironic-api
```

   Ubuntu:

```
sudo service ironic-api restart
```

Configuring ironic-api behind mod_wsgi

Bare Metal service comes with an example file for configuring the `ironic-api` service to run behind Apache with mod_wsgi.

Note: This is optional, the ironic APIs can be run using independent scripts that provide HTTP servers. But it is generally considered more performant and flexible to run them using a generic HTTP server that supports WSGI (such as Apache or nginx).

---

1. Install the apache service:

   Fedora/RHEL8/CentOS8:

   ```
   sudo dnf install httpd
   ```

   Debian/Ubuntu:

   ```
   apt-get install apache2
   ```

   SUSE:

   ```
   zypper install apache2
   ```

2. Download the `etc/apache2/ironic` file from the Ironic project tree and copy it to the apache sites:

   Fedora/RHEL8/CentOS8:

   ```
   sudo cp etc/apache2/ironic /etc/httpd/conf.d/ironic.conf
   ```

   Debian/Ubuntu:

   ```
   sudo cp etc/apache2/ironic /etc/apache2/sites-available/ironic.conf
   ```

   SUSE:

   ```
   sudo cp etc/apache2/ironic /etc/apache2/vhosts.d/ironic.conf
   ```

3. Edit the recently copied <`apache-configuration-dir`>`/ironic.conf`:

   1. Modify the `WSGIDaemonProcess`, `APACHE_RUN_USER` and `APACHE_RUN_GROUP` directives to set the user and group values to an appropriate user on your server.
   2. Modify the `WSGIScriptAlias` directive to point to the automatically generated `ironic-api-wsgi` script that is located in *IRONIC_BIN* directory.
   3. Modify the `Directory` directive to set the path to the Ironic API code.
   4. Modify the `ErrorLog` and `CustomLog` to redirect the logs to the right directory (on Red Hat systems this is usually under /var/log/httpd).

4. Stop and disable the ironic-api service. If ironic-api service is started, the port will be occupied. Apach will fail to start:

   Fedora/RHEL8/CentOS8/SUSE:

```
sudo systemctl stop openstack-ironic-api
sudo systemctl disable openstack-ironic-api
```

Debian/Ubuntu:

```
sudo service ironic-api stop
sudo service ironic-api disable
```

5. Enable the apache `ironic` in site and reload:

   Fedora/RHEL8/CentOS8:

   ```
   sudo systemctl reload httpd
   ```

   Debian/Ubuntu:

   ```
   sudo a2ensite ironic
   sudo service apache2 reload
   ```

   SUSE:

   ```
   sudo systemctl reload apache2
   ```

Note: The file `ironic-api-wsgi` is automatically generated by pbr and is available in *IRONIC_BIN* directory. It should not be modified.

Configure another WSGI container

A slightly different approach has to be used for WSGI containers that cannot use `ironic-api-wsgi`. For example, for *gunicorn*:

```
gunicorn -b 0.0.0.0:6385 'ironic.api.wsgi:initialize_wsgi_app(argv=[])'
```

If you want to pass a configuration file, use:

```
gunicorn -b 0.0.0.0:6385 \
    'ironic.api.wsgi:initialize_wsgi_app(argv=["ironic-api", "--config-file=/
↪path/to/_ironic.conf"])'
```

Configuring ironic-conductor service

1. Replace `HOST_IP` with IP of the conductor host.

   ```
   [DEFAULT]

   # IP address of this host. If unset, will determine the IP
   # programmatically. If unable to do so, will use "127.0.0.1".
   ```

(continues on next page)

(continued from previous page)

```
# (string value)
my_ip=HOST_IP
```

Note: If a conductor host has multiple IPs, `my_ip` should be set to the IP which is on the same network as the bare metal nodes.

2. Configure the location of the database. Ironic-conductor should use the same configuration as ironic-api. Replace `IRONIC_DBPASSWORD` with the password of your `ironic` user, and replace DB_IP with the IP address where the DB server is located:

```
[database]

# The SQLAlchemy connection string to use to connect to the
# database. (string value)
connection=mysql+pymysql://ironic:IRONIC_DBPASSWORD@DB_IP/ironic?
↪charset=utf8
```

3. Configure the ironic-conductor service to use the RabbitMQ message broker by setting the following option. Ironic-conductor should use the same configuration as ironic-api. Replace `RPC_*` with appropriate address details and credentials of RabbitMQ server:

```
[DEFAULT]

# A URL representing the messaging driver to use and its full
# configuration. (string value)
transport_url = rabbit://RPC_USER:RPC_PASSWORD@RPC_HOST:RPC_PORT/
```

Alternatively, you can use JSON RPC for interactions between ironic-conductor and ironic-api. Enable it in the configuration and provide the keystone credentials to use for authenticating incoming requests (can be the same as for the API):

```
[DEFAULT]

rpc_transport = json-rpc

[keystone_authtoken]

# Authentication type to load (string value)
auth_type=password

# Complete public Identity API endpoint (string value)
www_authenticate_uri=http://PUBLIC_IDENTITY_IP:5000

# Complete admin Identity API endpoint. (string value)
auth_url=http://PRIVATE_IDENTITY_IP:5000

# Service username. (string value)
username=ironic
```

(continues on next page)

(continued from previous page)

```
# Service account password. (string value)
password=IRONIC_PASSWORD

# Service tenant name. (string value)
project_name=service

# Domain name containing project (string value)
project_domain_name=Default

# User's domain name (string value)
user_domain_name=Default
```

You can optionally change the host and the port the JSON RPC service will bind to, for example:

```
[json_rpc]
host_ip = 192.168.0.10
port = 9999
```

Warning: Hostnames of ironic-conductor machines must be resolvable by ironic-api services when JSON RPC is used.

4. Configure credentials for accessing other OpenStack services.

   In order to communicate with other OpenStack services, the Bare Metal service needs to use service users to authenticate to the OpenStack Identity service when making requests to other services. These users credentials have to be configured in each configuration file section related to the corresponding service:

   - `[neutron]` - to access the OpenStack Networking service
   - `[glance]` - to access the OpenStack Image service
   - `[swift]` - to access the OpenStack Object Storage service
   - `[cinder]` - to access the OpenStack Block Storage service
   - `[inspector]` - to access the OpenStack Bare Metal Introspection service
   - `[service_catalog]` - a special section holding credentials the Bare Metal service will use to discover its own API URL endpoint as registered in the OpenStack Identity service catalog.

   For simplicity, you can use the same service user for all services. For backward compatibility, this should be the same user configured in the `[keystone_authtoken]` section for the ironic-api service (see Configuring ironic-api service). However, this is not necessary, and you can create and configure separate service users for each service.

   Under the hood, Bare Metal service uses `keystoneauth` library together with `Authentication plugin`, `Session` and `Adapter` concepts provided by it to instantiate service clients. Please refer to Keystoneauth documentation for supported plugins, their available options as well as Session- and Adapter-related options for authentication, connection and endpoint discovery respectively.

   In the example below, authentication information for user to access the OpenStack Networking service is configured to use:

- Networking service is deployed in the Identity service region named `RegionTwo`, with only its `public` endpoint interface registered in the service catalog.
- HTTPS connection with specific CA SSL certificate when making requests
- the same service user as configured for ironic-api service
- dynamic `password` authentication plugin that will discover appropriate version of Identity service API based on other provided options
  - replace `IDENTITY_IP` with the IP of the Identity server, and replace `IRONIC_PASSWORD` with the password you chose for the `ironic` user in the Identity service

```
[neutron]

# Authentication type to load (string value)
auth_type = password

# Authentication URL (string value)
auth_url=https://IDENTITY_IP:5000/

# Username (string value)
username=ironic

# User's password (string value)
password=IRONIC_PASSWORD

# Project name to scope to (string value)
project_name=service

# Domain ID containing project (string value)
project_domain_id=default

# User's domain id (string value)
user_domain_id=default

# PEM encoded Certificate Authority to use when verifying
# HTTPs connections. (string value)
cafile=/opt/stack/data/ca-bundle.pem

# The default region_name for endpoint URL discovery. (string
# value)
region_name = RegionTwo

# List of interfaces, in order of preference, for endpoint
# URL. (list value)
valid_interfaces=public
```

By default, in order to communicate with another service, the Bare Metal service will attempt to discover an appropriate endpoint for that service via the Identity services service catalog. The relevant configuration options from that service group in the Bare Metal service configuration file are used for this purpose. If you want to use a different endpoint for a particular service, specify this via the `endpoint_override` configuration option of that service group, in the Bare Metal

services configuration file. Taking the previous Networking service example, this would be

```
[neutron]
...
endpoint_override = <NEUTRON_API_ADDRESS>
```

(Replace *<NEUTRON_API_ADDRESS>* with actual address of a specific Networking service endpoint.)

5. Configure enabled drivers and hardware types as described in *Enabling drivers and hardware types*.

   A. If you enabled any driver that uses *Direct deploy*, Swift backend for the Image service must be installed and configured, see *Configure the Image service for temporary URLs*. Ceph Object Gateway (RADOS Gateway) is also supported as the Image services backend, see *Ceph Object Gateway support*.

6. Configure the network for ironic-conductor service to perform node cleaning, see *Node cleaning* from the admin guide.

7. Restart the ironic-conductor service:

   Fedora/RHEL7/CentOS7/SUSE:

   ```
   sudo systemctl restart openstack-ironic-conductor
   ```

   Ubuntu:

   ```
   sudo service ironic-conductor restart
   ```

Configuring single-process ironic

As an alternative to starting separate API and conductor instances, you can start `ironic` services that combine an API and a conductor in the same process. This may be particularly beneficial in environments with limited resources and low number of nodes to handle.

Note: This feature is available starting with the Yoga release series.

---

1. Start with setting up the environment as described in both *Configuring ironic-api service* and *Configuring ironic-conductor service*, but do not start any services. Merge configuration options into a single configuration file.

   Note: Any RPC settings will only take effect if you have more than one combined service started or if you have additional conductors.

   If you dont plan to have more than one conductor, you can disable the RPC completely:

   ```
   [DEFAULT]
   rpc_transport = none
   ```

   ---

2. Stop existing services if they are already started:

   Fedora/RHEL/CentOS/SUSE:

```
sudo systemctl stop openstack-ironic-api
sudo systemctl stop openstack-ironic-conductor
```

Ubuntu:

```
sudo service ironic-api stop
sudo service ironic-conductor stop
```

3. Start or restart the ironic service:

   Fedora/RHEL8/CentOS8/SUSE:

```
sudo systemctl restart openstack-ironic
```

   Ubuntu:

```
sudo service ironic restart
```

Install and configure for Ubuntu

This section describes how to install and configure the Bare Metal service for Ubuntu 14.04 (LTS).

Install and configure prerequisites

The Bare Metal service is a collection of components that provides support to manage and provision physical machines. You can configure these components to run on separate nodes or the same node. In this guide, the components run on one node, typically the Compute Services compute node.

It assumes that the Identity, Image, Compute, and Networking services have already been set up.

Set up the database for Bare Metal

The Bare Metal service stores information in a database. This guide uses the MySQL database that is used by other OpenStack services.

1. In MySQL, create an `ironic` database that is accessible by the `ironic` user. Replace `IRONIC_DBPASSWORD` with a suitable password:

```
# mysql -u root -p
mysql> CREATE DATABASE ironic CHARACTER SET utf8;
mysql> GRANT ALL PRIVILEGES ON ironic.* TO 'ironic'@'localhost' \
       IDENTIFIED BY 'IRONIC_DBPASSWORD';
mysql> GRANT ALL PRIVILEGES ON ironic.* TO 'ironic'@'%' \
       IDENTIFIED BY 'IRONIC_DBPASSWORD';
```

Install and configure components

1. Install from packages (using apt-get)

```
# apt-get install ironic-api ironic-conductor python3-ironicclient
```

2. Enable services

   Services are enabled by default on Ubuntu.

The Bare Metal service is configured via its configuration file. This file is typically located at `/etc/ironic/ironic.conf`.

Although some configuration options are mentioned here, it is recommended that you review all the *Sample Configuration File* so that the Bare Metal service is configured for your needs.

It is possible to set up an ironic-api and an ironic-conductor services on the same host or different hosts. Users also can add new ironic-conductor hosts to deal with an increasing number of bare metal nodes. But the additional ironic-conductor services should be at the same version as that of existing ironic-conductor services.

Configuring ironic-api service

1. The Bare Metal service stores information in a database. This guide uses the MySQL database that is used by other OpenStack services.

   Configure the location of the database via the `connection` option. In the following, replace `IRONIC_DBPASSWORD` with the password of your `ironic` user, and replace `DB_IP` with the IP address where the DB server is located:

```
[database]

# The SQLAlchemy connection string used to connect to the
# database (string value)
connection=mysql+pymysql://ironic:IRONIC_DBPASSWORD@DB_IP/ironic?
↪charset=utf8
```

2. Configure the ironic-api service to use the RabbitMQ message broker by setting the following option. Replace `RPC_*` with appropriate address details and credentials of RabbitMQ server:

```
[DEFAULT]

# A URL representing the messaging driver to use and its full
# configuration. (string value)
transport_url = rabbit://RPC_USER:RPC_PASSWORD@RPC_HOST:RPC_PORT/
```

   Alternatively, you can use JSON RPC for interactions between ironic-conductor and ironic-api. Enable it in the configuration and provide the keystone credentials to use for authentication:

```
[DEFAULT]

rpc_transport = json-rpc
```

(continues on next page)

(continued from previous page)

```
[json_rpc]

# Authentication type to load (string value)
auth_type = password

# Authentication URL (string value)
auth_url=https://IDENTITY_IP:5000/

# Username (string value)
username=ironic

# User's password (string value)
password=IRONIC_PASSWORD

# Project name to scope to (string value)
project_name=service

# Domain ID containing project (string value)
project_domain_id=default

# User's domain id (string value)
user_domain_id=default
```

If you use port other than the default 8089 for JSON RPC, you have to configure it, for example:

```
[json_rpc]
port = 9999
```

3. Configure the ironic-api service to use these credentials with the Identity service. Replace `PUBLIC_IDENTITY_IP` with the public IP of the Identity server, `PRIVATE_IDENTITY_IP` with the private IP of the Identity server and replace `IRONIC_PASSWORD` with the password you chose for the `ironic` user in the Identity service:

```
[DEFAULT]

# Authentication strategy used by ironic-api: one of
# "keystone" or "noauth". "noauth" should not be used in a
# production environment because all authentication will be
# disabled. (string value)
auth_strategy=keystone

[keystone_authtoken]

# Authentication type to load (string value)
auth_type=password

# Complete public Identity API endpoint (string value)
www_authenticate_uri=http://PUBLIC_IDENTITY_IP:5000

# Complete admin Identity API endpoint. (string value)
```

(continues on next page)

(continued from previous page)

```
auth_url=http://PRIVATE_IDENTITY_IP:5000

# Service username. (string value)
username=ironic

# Service account password. (string value)
password=IRONIC_PASSWORD

# Service tenant name. (string value)
project_name=service

# Domain name containing project (string value)
project_domain_name=Default

# User's domain name (string value)
user_domain_name=Default
```

4. Create the Bare Metal service database tables:

```
$ ironic-dbsync --config-file /etc/ironic/ironic.conf create_schema
```

5. Restart the ironic-api service:

   Fedora/RHEL8/CentOS8/SUSE:

```
sudo systemctl restart openstack-ironic-api
```

   Ubuntu:

```
sudo service ironic-api restart
```

Configuring ironic-api behind mod_wsgi

Bare Metal service comes with an example file for configuring the `ironic-api` service to run behind Apache with mod_wsgi.

Note: This is optional, the ironic APIs can be run using independent scripts that provide HTTP servers. But it is generally considered more performant and flexible to run them using a generic HTTP server that supports WSGI (such as Apache or nginx).

---

1. Install the apache service:

   Fedora/RHEL8/CentOS8:

```
sudo dnf install httpd
```

   Debian/Ubuntu:

```
apt-get install apache2
```

SUSE:

```
zypper install apache2
```

2. Download the `etc/apache2/ironic` file from the Ironic project tree and copy it to the apache sites:

   Fedora/RHEL8/CentOS8:

   ```
   sudo cp etc/apache2/ironic /etc/httpd/conf.d/ironic.conf
   ```

   Debian/Ubuntu:

   ```
   sudo cp etc/apache2/ironic /etc/apache2/sites-available/ironic.conf
   ```

   SUSE:

   ```
   sudo cp etc/apache2/ironic /etc/apache2/vhosts.d/ironic.conf
   ```

3. Edit the recently copied `<apache-configuration-dir>/ironic.conf`:

   1. Modify the `WSGIDaemonProcess`, `APACHE_RUN_USER` and `APACHE_RUN_GROUP` directives to set the user and group values to an appropriate user on your server.
   2. Modify the `WSGIScriptAlias` directive to point to the automatically generated `ironic-api-wsgi` script that is located in *IRONIC_BIN* directory.
   3. Modify the `Directory` directive to set the path to the Ironic API code.
   4. Modify the `ErrorLog` and `CustomLog` to redirect the logs to the right directory (on Red Hat systems this is usually under /var/log/httpd).

4. Stop and disable the ironic-api service. If ironic-api service is started, the port will be occupied. Apach will fail to start:

   Fedora/RHEL8/CentOS8/SUSE:

   ```
   sudo systemctl stop openstack-ironic-api
   sudo systemctl disable openstack-ironic-api
   ```

   Debian/Ubuntu:

   ```
   sudo service ironic-api stop
   sudo service ironic-api disable
   ```

5. Enable the apache `ironic` in site and reload:

   Fedora/RHEL8/CentOS8:

   ```
   sudo systemctl reload httpd
   ```

   Debian/Ubuntu:

   ```
   sudo a2ensite ironic
   sudo service apache2 reload
   ```

   SUSE:

```
sudo systemctl reload apache2
```

Note: The file `ironic-api-wsgi` is automatically generated by pbr and is available in *IRONIC_BIN* directory. It should not be modified.

Configure another WSGI container

A slightly different approach has to be used for WSGI containers that cannot use `ironic-api-wsgi`. For example, for *gunicorn*:

```
gunicorn -b 0.0.0.0:6385 'ironic.api.wsgi:initialize_wsgi_app(argv=[])'
```

If you want to pass a configuration file, use:

```
gunicorn -b 0.0.0.0:6385 \
    'ironic.api.wsgi:initialize_wsgi_app(argv=["ironic-api", "--config-file=/
↪path/to/_ironic.conf"])'
```

Configuring ironic-conductor service

1. Replace `HOST_IP` with IP of the conductor host.

```
[DEFAULT]

# IP address of this host. If unset, will determine the IP
# programmatically. If unable to do so, will use "127.0.0.1".
# (string value)
my_ip=HOST_IP
```

Note: If a conductor host has multiple IPs, `my_ip` should be set to the IP which is on the same network as the bare metal nodes.

2. Configure the location of the database. Ironic-conductor should use the same configuration as ironic-api. Replace `IRONIC_DBPASSWORD` with the password of your `ironic` user, and replace DB_IP with the IP address where the DB server is located:

```
[database]

# The SQLAlchemy connection string to use to connect to the
# database. (string value)
connection=mysql+pymysql://ironic:IRONIC_DBPASSWORD@DB_IP/ironic?
↪charset=utf8
```

3. Configure the ironic-conductor service to use the RabbitMQ message broker by setting the following option. Ironic-conductor should use the same configuration as ironic-api. Replace `RPC_*` with appropriate address details and credentials of RabbitMQ server:

```
[DEFAULT]

# A URL representing the messaging driver to use and its full
# configuration. (string value)
transport_url = rabbit://RPC_USER:RPC_PASSWORD@RPC_HOST:RPC_PORT/
```

Alternatively, you can use JSON RPC for interactions between ironic-conductor and ironic-api. Enable it in the configuration and provide the keystone credentials to use for authenticating incoming requests (can be the same as for the API):

```
[DEFAULT]

rpc_transport = json-rpc

[keystone_authtoken]

# Authentication type to load (string value)
auth_type=password

# Complete public Identity API endpoint (string value)
www_authenticate_uri=http://PUBLIC_IDENTITY_IP:5000

# Complete admin Identity API endpoint. (string value)
auth_url=http://PRIVATE_IDENTITY_IP:5000

# Service username. (string value)
username=ironic

# Service account password. (string value)
password=IRONIC_PASSWORD

# Service tenant name. (string value)
project_name=service

# Domain name containing project (string value)
project_domain_name=Default

# User's domain name (string value)
user_domain_name=Default
```

You can optionally change the host and the port the JSON RPC service will bind to, for example:

```
[json_rpc]
host_ip = 192.168.0.10
port = 9999
```

Warning: Hostnames of ironic-conductor machines must be resolvable by ironic-api services when JSON RPC is used.

4. Configure credentials for accessing other OpenStack services.

In order to communicate with other OpenStack services, the Bare Metal service needs to use service users to authenticate to the OpenStack Identity service when making requests to other services. These users credentials have to be configured in each configuration file section related to the corresponding service:

- `[neutron]` - to access the OpenStack Networking service
- `[glance]` - to access the OpenStack Image service
- `[swift]` - to access the OpenStack Object Storage service
- `[cinder]` - to access the OpenStack Block Storage service
- `[inspector]` - to access the OpenStack Bare Metal Introspection service
- `[service_catalog]` - a special section holding credentials the Bare Metal service will use to discover its own API URL endpoint as registered in the OpenStack Identity service catalog.

For simplicity, you can use the same service user for all services. For backward compatibility, this should be the same user configured in the `[keystone_authtoken]` section for the ironic-api service (see Configuring ironic-api service). However, this is not necessary, and you can create and configure separate service users for each service.

Under the hood, Bare Metal service uses `keystoneauth` library together with `Authentication plugin`, `Session` and `Adapter` concepts provided by it to instantiate service clients. Please refer to Keystoneauth documentation for supported plugins, their available options as well as Session- and Adapter-related options for authentication, connection and endpoint discovery respectively.

In the example below, authentication information for user to access the OpenStack Networking service is configured to use:

- Networking service is deployed in the Identity service region named `RegionTwo`, with only its `public` endpoint interface registered in the service catalog.
- HTTPS connection with specific CA SSL certificate when making requests
- the same service user as configured for ironic-api service
- dynamic `password` authentication plugin that will discover appropriate version of Identity service API based on other provided options
  - replace `IDENTITY_IP` with the IP of the Identity server, and replace `IRONIC_PASSWORD` with the password you chose for the `ironic` user in the Identity service

```
[neutron]

# Authentication type to load (string value)
auth_type = password

# Authentication URL (string value)
auth_url=https://IDENTITY_IP:5000/

# Username (string value)
username=ironic

# User's password (string value)
password=IRONIC_PASSWORD
```

(continues on next page)

(continued from previous page)

```
# Project name to scope to (string value)
project_name=service

# Domain ID containing project (string value)
project_domain_id=default

# User's domain id (string value)
user_domain_id=default

# PEM encoded Certificate Authority to use when verifying
# HTTPs connections. (string value)
cafile=/opt/stack/data/ca-bundle.pem

# The default region_name for endpoint URL discovery. (string
# value)
region_name = RegionTwo

# List of interfaces, in order of preference, for endpoint
# URL. (list value)
valid_interfaces=public
```

By default, in order to communicate with another service, the Bare Metal service will attempt to discover an appropriate endpoint for that service via the Identity services service catalog. The relevant configuration options from that service group in the Bare Metal service configuration file are used for this purpose. If you want to use a different endpoint for a particular service, specify this via the `endpoint_override` configuration option of that service group, in the Bare Metal services configuration file. Taking the previous Networking service example, this would be

```
[neutron]
...
endpoint_override = <NEUTRON_API_ADDRESS>
```

(Replace *<NEUTRON_API_ADDRESS>* with actual address of a specific Networking service endpoint.)

5. Configure enabled drivers and hardware types as described in *Enabling drivers and hardware types*.

   A. If you enabled any driver that uses *Direct deploy*, Swift backend for the Image service must be installed and configured, see *Configure the Image service for temporary URLs*. Ceph Object Gateway (RADOS Gateway) is also supported as the Image services backend, see *Ceph Object Gateway support*.

6. Configure the network for ironic-conductor service to perform node cleaning, see *Node cleaning* from the admin guide.

7. Restart the ironic-conductor service:

   Fedora/RHEL7/CentOS7/SUSE:

   ```
   sudo systemctl restart openstack-ironic-conductor
   ```

   Ubuntu:

```
sudo service ironic-conductor restart
```

Configuring single-process ironic

As an alternative to starting separate API and conductor instances, you can start `ironic` services that combine an API and a conductor in the same process. This may be particularly beneficial in environments with limited resources and low number of nodes to handle.

Note: This feature is available starting with the Yoga release series.

---

1. Start with setting up the environment as described in both *Configuring ironic-api service* and *Configuring ironic-conductor service*, but do not start any services. Merge configuration options into a single configuration file.

   Note: Any RPC settings will only take effect if you have more than one combined service started or if you have additional conductors.

   If you dont plan to have more than one conductor, you can disable the RPC completely:

   ```
   [DEFAULT]
   rpc_transport = none
   ```

   ---

2. Stop existing services if they are already started:

   Fedora/RHEL/CentOS/SUSE:

   ```
   sudo systemctl stop openstack-ironic-api
   sudo systemctl stop openstack-ironic-conductor
   ```

   Ubuntu:

   ```
   sudo service ironic-api stop
   sudo service ironic-conductor stop
   ```

3. Start or restart the ironic service:

   Fedora/RHEL8/CentOS8/SUSE:

   ```
   sudo systemctl restart openstack-ironic
   ```

   Ubuntu:

   ```
   sudo service ironic restart
   ```

Install and configure for openSUSE and SUSE Linux Enterprise

This section describes how to install and configure the Bare Metal service for openSUSE Leap 42.2 and SUSE Linux Enterprise Server 12 SP2.

Note: Installation of the Bare Metal service on openSUSE and SUSE Linux Enterprise Server is not officially supported. Nevertheless, installation should be possible.

Install and configure prerequisites

The Bare Metal service is a collection of components that provides support to manage and provision physical machines. You can configure these components to run on separate nodes or the same node. In this guide, the components run on one node, typically the Compute Services compute node.

It assumes that the Identity, Image, Compute, and Networking services have already been set up.

Set up the database for Bare Metal

The Bare Metal service stores information in a database. This guide uses the MySQL database that is used by other OpenStack services.

1. In MySQL, create an `ironic` database that is accessible by the `ironic` user. Replace `IRONIC_DBPASSWORD` with a suitable password:

```
# mysql -u root -p
mysql> CREATE DATABASE ironic CHARACTER SET utf8;
mysql> GRANT ALL PRIVILEGES ON ironic.* TO 'ironic'@'localhost' \
       IDENTIFIED BY 'IRONIC_DBPASSWORD';
mysql> GRANT ALL PRIVILEGES ON ironic.* TO 'ironic'@'%' \
       IDENTIFIED BY 'IRONIC_DBPASSWORD';
```

Install and configure components

1. Install from packages

```
# zypper install openstack-ironic-api openstack-ironic-conductor python3-
↪ironicclient
```

2. Enable services

```
# systemctl enable openstack-ironic-api openstack-ironic-conductor
# systemctl start openstack-ironic-api openstack-ironic-conductor
```

The Bare Metal service is configured via its configuration file. This file is typically located at `/etc/ironic/ironic.conf`.

Although some configuration options are mentioned here, it is recommended that you review all the *Sample Configuration File* so that the Bare Metal service is configured for your needs.

It is possible to set up an ironic-api and an ironic-conductor services on the same host or different hosts. Users also can add new ironic-conductor hosts to deal with an increasing number of bare metal nodes. But the additional ironic-conductor services should be at the same version as that of existing ironic-conductor services.

Configuring ironic-api service

1. The Bare Metal service stores information in a database. This guide uses the MySQL database that is used by other OpenStack services.

   Configure the location of the database via the `connection` option. In the following, replace `IRONIC_DBPASSWORD` with the password of your `ironic` user, and replace `DB_IP` with the IP address where the DB server is located:

   ```
   [database]

   # The SQLAlchemy connection string used to connect to the
   # database (string value)
   connection=mysql+pymysql://ironic:IRONIC_DBPASSWORD@DB_IP/ironic?
   ↪charset=utf8
   ```

2. Configure the ironic-api service to use the RabbitMQ message broker by setting the following option. Replace `RPC_*` with appropriate address details and credentials of RabbitMQ server:

   ```
   [DEFAULT]

   # A URL representing the messaging driver to use and its full
   # configuration. (string value)
   transport_url = rabbit://RPC_USER:RPC_PASSWORD@RPC_HOST:RPC_PORT/
   ```

   Alternatively, you can use JSON RPC for interactions between ironic-conductor and ironic-api. Enable it in the configuration and provide the keystone credentials to use for authentication:

   ```
   [DEFAULT]

   rpc_transport = json-rpc

   [json_rpc]

   # Authentication type to load (string value)
   auth_type = password

   # Authentication URL (string value)
   auth_url=https://IDENTITY_IP:5000/

   # Username (string value)
   username=ironic

   # User's password (string value)
   password=IRONIC_PASSWORD
   ```

(continues on next page)

(continued from previous page)

```
# Project name to scope to (string value)
project_name=service

# Domain ID containing project (string value)
project_domain_id=default

# User's domain id (string value)
user_domain_id=default
```

If you use port other than the default 8089 for JSON RPC, you have to configure it, for example:

```
[json_rpc]
port = 9999
```

3. Configure the ironic-api service to use these credentials with the Identity service. Replace `PUBLIC_IDENTITY_IP` with the public IP of the Identity server, `PRIVATE_IDENTITY_IP` with the private IP of the Identity server and replace `IRONIC_PASSWORD` with the password you chose for the `ironic` user in the Identity service:

```
[DEFAULT]

# Authentication strategy used by ironic-api: one of
# "keystone" or "noauth". "noauth" should not be used in a
# production environment because all authentication will be
# disabled. (string value)
auth_strategy=keystone

[keystone_authtoken]

# Authentication type to load (string value)
auth_type=password

# Complete public Identity API endpoint (string value)
www_authenticate_uri=http://PUBLIC_IDENTITY_IP:5000

# Complete admin Identity API endpoint. (string value)
auth_url=http://PRIVATE_IDENTITY_IP:5000

# Service username. (string value)
username=ironic

# Service account password. (string value)
password=IRONIC_PASSWORD

# Service tenant name. (string value)
project_name=service

# Domain name containing project (string value)
project_domain_name=Default
```

(continues on next page)

(continued from previous page)

```
# User's domain name (string value)
user_domain_name=Default
```

4. Create the Bare Metal service database tables:

```
$ ironic-dbsync --config-file /etc/ironic/ironic.conf create_schema
```

5. Restart the ironic-api service:

   Fedora/RHEL8/CentOS8/SUSE:

```
sudo systemctl restart openstack-ironic-api
```

   Ubuntu:

```
sudo service ironic-api restart
```

Configuring ironic-api behind mod_wsgi

Bare Metal service comes with an example file for configuring the `ironic-api` service to run behind Apache with mod_wsgi.

Note: This is optional, the ironic APIs can be run using independent scripts that provide HTTP servers. But it is generally considered more performant and flexible to run them using a generic HTTP server that supports WSGI (such as Apache or nginx).

---

1. Install the apache service:

   Fedora/RHEL8/CentOS8:

```
sudo dnf install httpd
```

   Debian/Ubuntu:

```
apt-get install apache2
```

   SUSE:

```
zypper install apache2
```

2. Download the `etc/apache2/ironic` file from the Ironic project tree and copy it to the apache sites:

   Fedora/RHEL8/CentOS8:

```
sudo cp etc/apache2/ironic /etc/httpd/conf.d/ironic.conf
```

   Debian/Ubuntu:

```
sudo cp etc/apache2/ironic /etc/apache2/sites-available/ironic.conf
```

SUSE:

```
sudo cp etc/apache2/ironic /etc/apache2/vhosts.d/ironic.conf
```

3. Edit the recently copied <apache-configuration-dir>/ironic.conf:
   1. Modify the WSGIDaemonProcess, APACHE_RUN_USER and APACHE_RUN_GROUP directives to set the user and group values to an appropriate user on your server.
   2. Modify the WSGIScriptAlias directive to point to the automatically generated ironic-api-wsgi script that is located in *IRONIC_BIN* directory.
   3. Modify the Directory directive to set the path to the Ironic API code.
   4. Modify the ErrorLog and CustomLog to redirect the logs to the right directory (on Red Hat systems this is usually under /var/log/httpd).
4. Stop and disable the ironic-api service. If ironic-api service is started, the port will be occupied. Apach will fail to start:

   Fedora/RHEL8/CentOS8/SUSE:

   ```
   sudo systemctl stop openstack-ironic-api
   sudo systemctl disable openstack-ironic-api
   ```

   Debian/Ubuntu:

   ```
   sudo service ironic-api stop
   sudo service ironic-api disable
   ```

5. Enable the apache ironic in site and reload:

   Fedora/RHEL8/CentOS8:

   ```
   sudo systemctl reload httpd
   ```

   Debian/Ubuntu:

   ```
   sudo a2ensite ironic
   sudo service apache2 reload
   ```

   SUSE:

   ```
   sudo systemctl reload apache2
   ```

Note: The file ironic-api-wsgi is automatically generated by pbr and is available in *IRONIC_BIN* directory. It should not be modified.

Configure another WSGI container

A slightly different approach has to be used for WSGI containers that cannot use `ironic-api-wsgi`. For example, for *gunicorn*:

```
gunicorn -b 0.0.0.0:6385 'ironic.api.wsgi:initialize_wsgi_app(argv=[])'
```

If you want to pass a configuration file, use:

```
gunicorn -b 0.0.0.0:6385 \
    'ironic.api.wsgi:initialize_wsgi_app(argv=["ironic-api", "--config-file=/
↪path/to/_ironic.conf"])'
```

Configuring ironic-conductor service

1. Replace `HOST_IP` with IP of the conductor host.

   ```
   [DEFAULT]

   # IP address of this host. If unset, will determine the IP
   # programmatically. If unable to do so, will use "127.0.0.1".
   # (string value)
   my_ip=HOST_IP
   ```

   Note: If a conductor host has multiple IPs, `my_ip` should be set to the IP which is on the same network as the bare metal nodes.

2. Configure the location of the database. Ironic-conductor should use the same configuration as ironic-api. Replace `IRONIC_DBPASSWORD` with the password of your `ironic` user, and replace DB_IP with the IP address where the DB server is located:

   ```
   [database]

   # The SQLAlchemy connection string to use to connect to the
   # database. (string value)
   connection=mysql+pymysql://ironic:IRONIC_DBPASSWORD@DB_IP/ironic?
   ↪charset=utf8
   ```

3. Configure the ironic-conductor service to use the RabbitMQ message broker by setting the following option. Ironic-conductor should use the same configuration as ironic-api. Replace `RPC_*` with appropriate address details and credentials of RabbitMQ server:

   ```
   [DEFAULT]

   # A URL representing the messaging driver to use and its full
   # configuration. (string value)
   transport_url = rabbit://RPC_USER:RPC_PASSWORD@RPC_HOST:RPC_PORT/
   ```

Alternatively, you can use JSON RPC for interactions between ironic-conductor and ironic-api. Enable it in the configuration and provide the keystone credentials to use for authenticating incoming requests (can be the same as for the API):

```
[DEFAULT]

rpc_transport = json-rpc

[keystone_authtoken]

# Authentication type to load (string value)
auth_type=password

# Complete public Identity API endpoint (string value)
www_authenticate_uri=http://PUBLIC_IDENTITY_IP:5000

# Complete admin Identity API endpoint. (string value)
auth_url=http://PRIVATE_IDENTITY_IP:5000

# Service username. (string value)
username=ironic

# Service account password. (string value)
password=IRONIC_PASSWORD

# Service tenant name. (string value)
project_name=service

# Domain name containing project (string value)
project_domain_name=Default

# User's domain name (string value)
user_domain_name=Default
```

You can optionally change the host and the port the JSON RPC service will bind to, for example:

```
[json_rpc]
host_ip = 192.168.0.10
port = 9999
```

Warning: Hostnames of ironic-conductor machines must be resolvable by ironic-api services when JSON RPC is used.

4. Configure credentials for accessing other OpenStack services.

   In order to communicate with other OpenStack services, the Bare Metal service needs to use service users to authenticate to the OpenStack Identity service when making requests to other services. These users credentials have to be configured in each configuration file section related to the corresponding service:

   - `[neutron]` - to access the OpenStack Networking service

- `[glance]` - to access the OpenStack Image service
- `[swift]` - to access the OpenStack Object Storage service
- `[cinder]` - to access the OpenStack Block Storage service
- `[inspector]` - to access the OpenStack Bare Metal Introspection service
- `[service_catalog]` - a special section holding credentials the Bare Metal service will use to discover its own API URL endpoint as registered in the OpenStack Identity service catalog.

For simplicity, you can use the same service user for all services. For backward compatibility, this should be the same user configured in the `[keystone_authtoken]` section for the ironic-api service (see Configuring ironic-api service). However, this is not necessary, and you can create and configure separate service users for each service.

Under the hood, Bare Metal service uses `keystoneauth` library together with `Authentication plugin`, `Session` and `Adapter` concepts provided by it to instantiate service clients. Please refer to Keystoneauth documentation for supported plugins, their available options as well as Session- and Adapter-related options for authentication, connection and endpoint discovery respectively.

In the example below, authentication information for user to access the OpenStack Networking service is configured to use:

- Networking service is deployed in the Identity service region named `RegionTwo`, with only its `public` endpoint interface registered in the service catalog.
- HTTPS connection with specific CA SSL certificate when making requests
- the same service user as configured for ironic-api service
- dynamic `password` authentication plugin that will discover appropriate version of Identity service API based on other provided options
  - replace `IDENTITY_IP` with the IP of the Identity server, and replace `IRONIC_PASSWORD` with the password you chose for the `ironic` user in the Identity service

```
[neutron]

# Authentication type to load (string value)
auth_type = password

# Authentication URL (string value)
auth_url=https://IDENTITY_IP:5000/

# Username (string value)
username=ironic

# User's password (string value)
password=IRONIC_PASSWORD

# Project name to scope to (string value)
project_name=service

# Domain ID containing project (string value)
project_domain_id=default
```

(continues on next page)

(continued from previous page)

```
# User's domain id (string value)
user_domain_id=default

# PEM encoded Certificate Authority to use when verifying
# HTTPs connections. (string value)
cafile=/opt/stack/data/ca-bundle.pem

# The default region_name for endpoint URL discovery. (string
# value)
region_name = RegionTwo

# List of interfaces, in order of preference, for endpoint
# URL. (list value)
valid_interfaces=public
```

By default, in order to communicate with another service, the Bare Metal service will attempt to discover an appropriate endpoint for that service via the Identity services service catalog. The relevant configuration options from that service group in the Bare Metal service configuration file are used for this purpose. If you want to use a different endpoint for a particular service, specify this via the `endpoint_override` configuration option of that service group, in the Bare Metal services configuration file. Taking the previous Networking service example, this would be

```
[neutron]
...
endpoint_override = <NEUTRON_API_ADDRESS>
```

(Replace *<NEUTRON_API_ADDRESS>* with actual address of a specific Networking service endpoint.)

5. Configure enabled drivers and hardware types as described in *Enabling drivers and hardware types*.

   A. If you enabled any driver that uses *Direct deploy*, Swift backend for the Image service must be installed and configured, see *Configure the Image service for temporary URLs*. Ceph Object Gateway (RADOS Gateway) is also supported as the Image services backend, see *Ceph Object Gateway support*.

6. Configure the network for ironic-conductor service to perform node cleaning, see *Node cleaning* from the admin guide.

7. Restart the ironic-conductor service:

   Fedora/RHEL7/CentOS7/SUSE:

   ```
   sudo systemctl restart openstack-ironic-conductor
   ```

   Ubuntu:

   ```
   sudo service ironic-conductor restart
   ```

Configuring single-process ironic

As an alternative to starting separate API and conductor instances, you can start `ironic` services that combine an API and a conductor in the same process. This may be particularly beneficial in environments with limited resources and low number of nodes to handle.

Note: This feature is available starting with the Yoga release series.

---

1. Start with setting up the environment as described in both *Configuring ironic-api service* and *Configuring ironic-conductor service*, but do not start any services. Merge configuration options into a single configuration file.

   Note: Any RPC settings will only take effect if you have more than one combined service started or if you have additional conductors.

   If you dont plan to have more than one conductor, you can disable the RPC completely:

   ```
   [DEFAULT]
   rpc_transport = none
   ```

   ---

2. Stop existing services if they are already started:

   Fedora/RHEL/CentOS/SUSE:

   ```
   sudo systemctl stop openstack-ironic-api
   sudo systemctl stop openstack-ironic-conductor
   ```

   Ubuntu:

   ```
   sudo service ironic-api stop
   sudo service ironic-conductor stop
   ```

3. Start or restart the ironic service:

   Fedora/RHEL8/CentOS8/SUSE:

   ```
   sudo systemctl restart openstack-ironic
   ```

   Ubuntu:

   ```
   sudo service ironic restart
   ```

2.1.4 Building or downloading a deploy ramdisk image

Ironic depends on having an image with the ironic-python-agent (IPA) service running on it for controlling and deploying bare metal nodes.

Two kinds of images are published on every commit from every branch of ironic-python-agent (IPA)

- DIB images are suitable for production usage and can be downloaded from https://tarballs.openstack.org/ironic-python-agent/dib/files/.
  - For Train and older use CentOS 7 images.
  - For Ussuri and newer use CentOS 8 images.

> **Warning:** CentOS 7 master images are no longer updated and must not be used.

> **Warning:** The published images will not work for dhcp-less deployments since the simple-init element is not present. Check the DIB documentation to see how to build the image.

- TinyIPA images are suitable for CI and testing environments and can be downloaded from https://tarballs.openstack.org/ironic-python-agent/tinyipa/files/.

Building from source

Check the ironic-python-agent-builder project for information on how to build ironic-python-agent ramdisks.

2.1.5 Integration with other OpenStack services

Configure the Identity service for the Bare Metal service

1. Create the Bare Metal service user (for example, `ironic`). The service uses this to authenticate with the Identity service. Use the `service` tenant and give the user the `admin` role:

```
$ openstack user create --password IRONIC_PASSWORD \
    --email ironic@example.com ironic
$ openstack role add --project service --user ironic admin
```

2. You must register the Bare Metal service with the Identity service so that other OpenStack services can locate it. To register the service:

```
$ openstack service create --name ironic --description \
    "Ironic baremetal provisioning service" baremetal
```

3. Use the `id` property that is returned from the Identity service when registering the service (above), to create the endpoint, and replace `IRONIC_NODE` with your Bare Metal services API node:

```
$ openstack endpoint create --region RegionOne \
    baremetal admin http://$IRONIC_NODE:6385
$ openstack endpoint create --region RegionOne \
    baremetal public http://$IRONIC_NODE:6385
$ openstack endpoint create --region RegionOne \
    baremetal internal http://$IRONIC_NODE:6385
```

4. You may delegate limited privileges related to the Bare Metal service to your Users by creating Roles with the OpenStack Identity service. By default, the Bare Metal service expects the baremetal_admin and baremetal_observer Roles to exist, in addition to the default admin Role. There is no negative consequence if you choose not to create these Roles. They can be created with the following commands:

   ```
   $ openstack role create baremetal_admin
   $ openstack role create baremetal_observer
   ```

   If you choose to customize the names of Roles used with the Bare Metal service, do so by changing the is_member, is_observer, and is_admin policy settings in `/etc/ironic/policy.yaml`.

   More complete documentation on managing Users and Roles within your OpenStack deployment are outside the scope of this document, but may be found here.

5. You can further restrict access to the Bare Metal service by creating a separate baremetal Project, so that Bare Metal resources (Nodes, Ports, etc) are only accessible to members of this Project:

   ```
   $ openstack project create baremetal
   ```

   At this point, you may grant read-only access to the Bare Metal service API without granting any other access by issuing the following commands:

   ```
   $ openstack user create \
     --domain default --project-domain default --project baremetal \
     --password PASSWORD USERNAME
   $ openstack role add \
     --user-domain default --project-domain default --project baremetal \
     --user USERNAME baremetal_observer
   ```

6. Further documentation is available elsewhere for the `openstack` command-line client and the Identity service. A *policy.yaml.sample* file, which enumerates the services default policies, is provided for your convenience with the Bare Metal Service.

Configure the Compute service to use the Bare Metal service

The Compute service needs to be configured to use the Bare Metal services driver. The configuration file for the Compute service is typically located at `/etc/nova/nova.conf`.

Note: As of the Newton release, it is possible to have multiple nova-compute services running the ironic virtual driver (in nova) to provide redundancy. Bare metal nodes are mapped to the services via a hash ring. If a service goes down, the available bare metal nodes are remapped to different services.

Once active, a node will stay mapped to the same nova-compute even when it goes down. The node is unable to be managed through the Compute API until the service responsible returns to an active state.

The following configuration file must be modified on the Compute services controller nodes and compute nodes.

1. Change these configuration options in the Compute service configuration file (for example, `/etc/nova/nova.conf`):

```
[default]

# Defines which driver to use for controlling virtualization.
# Enable the ironic virt driver for this compute instance.
compute_driver=ironic.IronicDriver

# Amount of memory in MB to reserve for the host so that it is always
# available to host processes.
# It is impossible to reserve any memory on bare metal nodes, so set
# this to zero.
reserved_host_memory_mb=0

[filter_scheduler]

# Enables querying of individual hosts for instance information.
# Not possible for bare metal nodes, so set it to False.
track_instance_changes=False

[scheduler]

# This value controls how often (in seconds) the scheduler should
# attempt to discover new hosts that have been added to cells.
# If negative (the default), no automatic discovery will occur.
# As each bare metal node is represented by a separate host, it has
# to be discovered before the Compute service can deploy on it.
# The value here has to be carefully chosen based on a compromise
# between the enrollment speed and the load on the Compute scheduler.
# The recommended value of 2 minutes matches how often the Compute
# service polls the Bare Metal service for node information.
discover_hosts_in_cells_interval=120
```

Note: The alternative to setting the `discover_hosts_in_cells_interval` option is to run the following command on any Compute controller node after each node is enrolled:

```
nova-manage cell_v2 discover_hosts --by-service
```

2. Consider enabling the following option on controller nodes:

```
[filter_scheduler]
```

(continues on next page)

(continued from previous page)

```
# Enabling this option is beneficial as it reduces re-scheduling events
# for ironic nodes when scheduling is based on resource classes,
# especially for mixed hypervisor case with host_subset_size = 1.
# However enabling it will also make packing of VMs on hypervisors
# less dense even when scheduling weights are completely disabled.
#shuffle_best_same_weighed_hosts = false
```

3. Carefully consider the following option:

```
[compute]

# This option will cause nova-compute to set itself to a disabled state
# if a certain number of consecutive build failures occur. This will
# prevent the scheduler from continuing to send builds to a compute
# service that is consistently failing. In the case of bare metal
# provisioning, however, a compute service is rarely the cause of build
# failures. Furthermore, bare metal nodes, managed by a disabled
# compute service, will be remapped to a different one. That may cause
# the second compute service to also be disabled, and so on, until no
# compute services are active.
# If this is not the desired behavior, consider increasing this value or
# setting it to 0 to disable this behavior completely.
#consecutive_build_service_disable_threshold = 10
```

4. Change these configuration options in the `ironic` section. Replace:

   - `IRONIC_PASSWORD` with the password you chose for the `ironic` user in the Identity Service
   - `IRONIC_NODE` with the hostname or IP address of the ironic-api node
   - `IDENTITY_IP` with the IP of the Identity server

```
[ironic]

# Ironic authentication type
auth_type=password

# Keystone API endpoint
auth_url=http://IDENTITY_IP:5000/v3

# Ironic keystone project name
project_name=service

# Ironic keystone admin name
username=ironic

# Ironic keystone admin password
password=IRONIC_PASSWORD

# Ironic keystone project domain
# or set project_domain_id
```

(continues on next page)

(continued from previous page)

```
project_domain_name=Default

# Ironic keystone user domain
# or set user_domain_id
user_domain_name=Default
```

5. On the Compute services controller nodes, restart the `nova-scheduler` process:

```
Fedora/RHEL8/CentOS8/SUSE:
  sudo systemctl restart openstack-nova-scheduler

Ubuntu:
  sudo service nova-scheduler restart
```

6. On the Compute services compute nodes, restart the `nova-compute` process:

```
Fedora/RHEL8/CentOS8/SUSE:
  sudo systemctl restart openstack-nova-compute

Ubuntu:
  sudo service nova-compute restart
```

Configure the Networking service for bare metal provisioning

You need to configure Networking so that the bare metal server can communicate with the Networking service for DHCP, PXE boot and other requirements. This section covers configuring Networking for a single flat network for bare metal provisioning.

It is recommended to use the baremetal ML2 mechanism driver and L2 agent for proper integration with the Networking service. Documentation regarding installation and configuration of the baremetal mechanism driver and L2 agent is available here.

For use with routed networks the baremetal ML2 components are required.

Note: When the baremetal ML2 components are *not* used, ports in the Networking service will have status: `DOWN`, and binding_vif_type: `binding_failed`. This was always the status for Bare Metal service `flat` network interface ports prior to the introduction of the baremetal ML2 integration. For a non-routed network, bare metal servers can still be deployed and are functional, despite this port binding state in the Networking service.

You will also need to provide Bare Metal service with the MAC address(es) of each node that it is provisioning; Bare Metal service in turn will pass this information to Networking service for DHCP and PXE boot configuration. An example of this is shown in the *Enrollment* section.

1. Install the networking-baremetal ML2 mechanism driver and L2 agent in the Networking service.
2. Edit `/etc/neutron/plugins/ml2/ml2_conf.ini` and modify these:

```
[ml2]
type_drivers = flat
```

(continues on next page)

(continued from previous page)

```
tenant_network_types = flat
mechanism_drivers = openvswitch,baremetal

[ml2_type_flat]
flat_networks = physnet1

[securitygroup]
firewall_driver = neutron.agent.linux.iptables_firewall.
↪OVSHybridIptablesFirewallDriver
enable_security_group = True

[ovs]
bridge_mappings = physnet1:br-eth2
# Replace eth2 with the interface on the neutron node which you
# are using to connect to the bare metal server
```

3. Restart the `neutron-server` service, to load the new configuration.
4. Create and edit `/etc/neutron/plugins/ml2/ironic_neutron_agent.ini` and add the required configuration. For example:

```
[ironic]
project_domain_name = Default
project_name = service
user_domain_name = Default
password = password
username = ironic
auth_url = http://identity-server.example.com/identity
auth_type = password
region_name = RegionOne
```

5. Make sure the `ironic-neutron-agent` service is started.
6. If neutron-openvswitch-agent runs with `ovs_neutron_plugin.ini` as the input config-file, edit `ovs_neutron_plugin.ini` to configure the bridge mappings by adding the [ovs] section described in the previous step, and restart the neutron-openvswitch-agent.
7. Add the integration bridge to Open vSwitch:

```
$ ovs-vsctl add-br br-int
```

8. Create the br-eth2 network bridge to handle communication between the OpenStack services (and the Bare Metal services) and the bare metal nodes using eth2. Replace eth2 with the interface on the network node which you are using to connect to the Bare Metal service:

```
$ ovs-vsctl add-br br-eth2
$ ovs-vsctl add-port br-eth2 eth2
```

9. Restart the Open vSwitch agent:

```
# service neutron-plugin-openvswitch-agent restart
```

10. On restarting the Networking service Open vSwitch agent, the veth pair between the bridges br-int and br-eth2 is automatically created.

    Your Open vSwitch bridges should look something like this after following the above steps:

```
$ ovs-vsctl show

    Bridge br-int
        fail_mode: secure
        Port "int-br-eth2"
            Interface "int-br-eth2"
                type: patch
                options: {peer="phy-br-eth2"}
        Port br-int
            Interface br-int
                type: internal
    Bridge "br-eth2"
        Port "phy-br-eth2"
            Interface "phy-br-eth2"
                type: patch
                options: {peer="int-br-eth2"}
        Port "eth2"
            Interface "eth2"
        Port "br-eth2"
            Interface "br-eth2"
                type: internal
    ovs_version: "2.3.0"
```

11. Create the flat network on which you are going to launch the instances:

```
$ openstack network create --project $TENANT_ID sharednet1 --share \
    --provider-network-type flat --provider-physical-network physnet1
```

12. Create the subnet on the newly created network:

```
$ openstack subnet create $SUBNET_NAME --network sharednet1 \
  --subnet-range $NETWORK_CIDR --ip-version 4 --gateway $GATEWAY_IP \
  --allocation-pool start=$START_IP,end=$END_IP --dhcp
```

Configuring services for bare metal provisioning using IPv6

Use of IPv6 addressing for baremetal provisioning requires additional configuration. This page covers the IPv6 specifics only. Please refer to *Configure tenant networks* and *Configure the Networking service for bare metal provisioning* for general networking configuration.

Configure ironic PXE driver for provisioning using IPv6 addressing

The PXE drivers operate in such a way that they are able to utilize both IPv4 and IPv6 addresses based upon the deployments operating state and configuration. Internally, the drivers attempt to prepare configuration options for both formats, which allows ports which are IPv6 only to automatically receieve boot parameters. As a result of this, it is critical that the `[DEFAULT]my_ipv6` configuration parameter is set to the conductors IPv6 address. This option is unique per conductor, and due to the nature of automatic address assignment, it cannot be guessed by the software.

Provisioning with IPv6 stateless addressing

When using stateless addressing DHCPv6 does not provide addresses to the client. DHCPv6 however provides other configuration via DHCPv6 options such as the bootfile-url and bootfile-parameters.

Once the PXE driver is set to operate in IPv6 mode no further configuration is required in the Baremetal Service.

Creating networks and subnets in the Networking Service

When creating the Baremetal Service network(s) and subnet(s) in the Networking Services, subnets should have `ipv6-address-mode` set to `dhcpv6-stateless` and `ip-version` set to `6`. Depending on whether a router in the Networking Service is providing RAs (Router Advertisements) or not, the `ipv6-ra-mode` for the subnet(s) should either be set to `dhcpv6-stateless` or be left unset.

Note: If `ipv6-ra-mode` is left unset, an external router on the network is expected to provide RAs with the appropriate flags set for automatic addressing and other configuration.

Provisioning with IPv6 stateful addressing

When using stateful addressing DHCPv6 is providing both addresses and other configuration via DHCPv6 options such as the bootfile-url and bootfile- parameters.

The identity-association (IA) construct used by DHCPv6 is challenging when booting over the network. Firmware, and ramdisks typically end up using different DUID/IAID combinations and it is not always possible for one chain- booting stage to release its address before giving control to the next step. In case the DHCPv6 server is configured with static reservations only the result is that booting will fail because the DHCPv6 server has no addresses available. To get past this issue either configure the DHCPv6 server with multiple address reservations for each host, or use a dynamic range.

Note: Support for multiple address reservations requires dnsmasq version 2.81 or later. Some distributions may backport this feature to earlier dnsmasq version as part of the packaging, check the distributions release notes.

If a different (not dnsmasq) DHCPv6 server backend is used with the Networking service, use of multiple address reservations might not work.

Using the `flat` network interface

Due to the identity-association challenges with DHCPv6 provisioning using the `flat` network interface is not recommended. When ironic operates with the `flat` network interface the server instance port is used for provisioning and other operations. Ironic will not use multiple address reservations in this scenario. Because of this **it will not work in most cases**.

Using the `neutron` network interface

When using the `neutron` network interface the Baremetal Service will allocate multiple IPv6 addresses (4 addresses per port by default) on the service networks used for provisioning, cleaning, rescue and introspection. The number of addresses allocated can be controlled via the `[neutron]/dhcpv6_stateful_address_count` option in the Bare Metal Services configuration file (`/etc/ironic/ironic.conf`). Using multiple address reservations ensures that the DHCPv6 server can lease addresses to each step.

To enable IPv6 provisioning on neutron *flat* provider networks with no switch management, the `local_link_connection` field of baremetal ports must be set to `{'network_type': 'unmanaged'}`. The following example shows how to set the local_link_connection for operation on unmanaged networks:

```
baremetal port set \
  --local-link-connection network_type=unmanaged <port-uuid>
```

The use of multiple IPv6 addresses must also be enabled in the Networking Services dhcp agent configuration (`/etc/neutron/dhcp_agent.ini`) by setting the option `[DEFAULT]/dnsmasq_enable_addr6_list` to `True` (default `False` in Ussuri release).

Note: Support for multiple IPv6 address reservations in the dnsmasq backend was added to the Networking Service Ussuri release. It was also backported to the stable Train release.

Creating networks and subnets in the Networking Service

When creating the ironic service network(s) and subnet(s) in the Networking Service, subnets should have `ipv6-address-mode` set to `dhcpv6-stateful` and `ip-version` set to `6`. Depending on whether a router in the Networking Service is providing RAs (Router Advertisements) or not, the `ipv6-ra-mode` for the subnet(s) should be set to either `dhcpv6-stateful` or be left unset.

Note: If `ipv6-ra-mode` is left unset, an external router on the network is expected to provide RAs with the appropriate flags set for managed addressing and other configuration.

Configure the Image service for temporary URLs

Some drivers of the Baremetal service (in particular, any drivers using *Direct deploy* or *Ansible deploy* interfaces, and some virtual media drivers) require target user images to be available over clean HTTP(S) URL with no authentication involved (neither username/password-based, nor token-based).

When using the Baremetal service integrated in OpenStack, this can be achieved by specific configuration of the Image service and Object Storage service as described below.

1. Configure the Image service to have object storage as a backend for storing images. For more details, please refer to the Image service configuration guide.

   Note: When using Ceph+RadosGW for Object Storage service, images stored in Image service must be available over Object Storage service as well.

   ---

2. Enable TempURLs for the Object Storage account used by the Image service for storing images in the Object Storage service.

   1. Check if TempURLs are enabled:

      ```
      # executed under credentials of the user used by Image service
      # to access Object Storage service
      $ openstack object store account show
      +------------+----------------------------------------+
      | Field      | Value                                  |
      +------------+----------------------------------------+
      | Account    | AUTH_bc39f1d9dcf9486899088007789ae643  |
      | Bytes      | 536661727                              |
      | Containers | 1                                      |
      | Objects    | 19                                     |
      | properties | Temp-Url-Key='secret'                  |
      +------------+----------------------------------------+
      ```

   2. If property `Temp-Url-Key` is set, note its value.

   3. If property `Temp-Url-Key` is not set, you have to configure it (`secret` is used in the example below for the value):

      ```
      $ openstack object store account set --property Temp-Url-Key=secret
      ```

3. Optionally, configure the ironic-conductor service. The default configuration assumes that:
   1. the Object Storage service is implemented by swift,
   2. the Object Storage service URL is available from the service catalog,
   3. the project, used by the Image service to access the Object Storage, is the same as the project, used by the Bare Metal service to access it,
   4. the container, used by the Image service, is called `glance`.

   If any of these assumptions do not hold, you may want to change your configuration file (typically located at `/etc/ironic/ironic.conf`), for example:

```
[glance]

swift_endpoint_url = http://openstack/swift
swift_account = AUTH_bc39f1d9dcf9486899088007789ae643
swift_container = glance
swift_temp_url_key = secret
```

4. (Re)start the ironic-conductor service.

Enabling HTTPS

Enabling HTTPS in Swift

The drivers using virtual media use swift for storing boot images and node configuration information (contains sensitive information for Ironic conductor to provision bare metal hardware). By default, HTTPS is not enabled in swift. HTTPS is required to encrypt all communication between swift and Ironic conductor and swift and bare metal (via virtual media). It can be enabled in one of the following ways:

- Using an SSL termination proxy
- Using native SSL support in swift (recommended only for testing purpose by swift).

Enabling HTTPS in Image service

Ironic drivers usually use Image service during node provisioning. By default, image service does not use HTTPS, but it is required for secure communication. It can be enabled by making the following changes to `/etc/glance/glance-api.conf`:

1. Configuring SSL support
2. Restart the glance-api service:

```
Fedora/RHEL8/CentOS8/SUSE:
    sudo systemctl restart openstack-glance-api

Debian/Ubuntu:
    sudo service glance-api restart
```

See the Glance documentation, for more details on the Image service.

Enabling HTTPS communication between Image service and Object storage

This section describes the steps needed to enable secure HTTPS communication between Image service and Object storage when Object storage is used as the Backend.

To enable secure HTTPS communication between Image service and Object storage follow these steps:

1. *Enabling HTTPS in Swift*
2. Configure Swift Storage Backend
3. *Enabling HTTPS in Image service*

Enabling HTTPS communication between Image service and Bare Metal service

This section describes the steps needed to enable secure HTTPS communication between Image service and Bare Metal service.

To enable secure HTTPS communication between Bare Metal service and Image service follow these steps:

1. Edit `/etc/ironic/ironic.conf`:

```
[glance]
...
glance_cafile=/path/to/certfile
```

Note: glance_cafile is an optional path to a CA certificate bundle to be used to validate the SSL certificate served by Image service.

2. If not using the keystone service catalog for the Image service API endpoint discovery, also edit the `endpoint_override` option to point to HTTPS URL of image service (replace `<GLANCE_API_ADDRESS>` with hostname[:port][path] of the Image service endpoint):

```
[glance]
...
endpoint_override = https://<GLANCE_API_ADDRESS>
```

3. Restart ironic-conductor service:

```
Fedora/RHEL8/CentOS8/SUSE:
    sudo systemctl restart openstack-ironic-conductor

Debian/Ubuntu:
    sudo service ironic-conductor restart
```

Configure the Bare Metal service for cleaning

Note: If you configured the Bare Metal service to do *Automated cleaning* (which is enabled by default), you will need to set the `cleaning_network` configuration option.

1. Note the network UUID (the *id* field) of the network you created in *Configure the Networking service for bare metal provisioning* or another network you created for cleaning:

```
$ openstack network list
```

2. Configure the cleaning network UUID via the `cleaning_network` option in the Bare Metal service configuration file (`/etc/ironic/ironic.conf`). In the following, replace `NETWORK_UUID` with the UUID you noted in the previous step:

```
[neutron]
cleaning_network = NETWORK_UUID
```

3. Restart the Bare Metal services ironic-conductor:

```
Fedora/RHEL8/CentOS8/SUSE:
  sudo systemctl restart openstack-ironic-conductor

Ubuntu:
  sudo service ironic-conductor restart
```

Configure tenant networks

Below is an example flow of how to set up the Bare Metal service so that node provisioning will happen in a multi-tenant environment (which means using the `neutron` network interface as stated above):

1. Network interfaces can be enabled on ironic-conductor by adding them to the `enabled_network_interfaces` configuration option under the `default` section of the configuration file:

```
[DEFAULT]
...
enabled_network_interfaces=noop,flat,neutron
```

   Keep in mind that, ideally, all ironic-conductors should have the same list of enabled network interfaces, but it may not be the case during ironic-conductor upgrades. This may cause problems if one of the ironic-conductors dies and some node that is taken over is mapped to an ironic-conductor that does not support the nodes network interface. Any actions that involve calling the nodes driver will fail until that network interface is installed and enabled on that ironic-conductor.

2. It is recommended to set the default network interface via the `default_network_interface` configuration option under the `default` section of the configuration file:

```
[DEFAULT]
...
default_network_interface=neutron
```

This default value will be used for all nodes that dont have a network interface explicitly specified in the creation request.

If this configuration option is not set, the default network interface is determined by looking at the `[dhcp]dhcp_provider` configuration option value. If it is `neutron`, then `flat` network interface becomes the default, otherwise `noop` is the default.

3. Define a provider network in the Networking service, which we shall refer to as the provisioning network. Using the `neutron` network interface requires that `provisioning_network` and `cleaning_network` configuration options are set to valid identifiers (UUID or name) of networks in the Networking service. If these options are not set correctly, cleaning or provisioning will fail to start. There are two ways to set these values:

   - Under the `neutron` section of ironic configuration file:

   ```
   [neutron]
   cleaning_network = $CLEAN_UUID_OR_NAME
   provisioning_network = $PROVISION_UUID_OR_NAME
   ```

   - Under `provisioning_network` and `cleaning_network` keys of the nodes `driver_info` field as `driver_info['provisioning_network']` and `driver_info['cleaning_network']` respectively.

Note: If these `provisioning_network` and `cleaning_network` values are not specified in nodes *driver_info* then ironic falls back to the configuration in the `neutron` section.

Please refer to *Configure the Bare Metal service for cleaning* for more information about cleaning.

Warning: Please make sure that the Bare Metal service has exclusive access to the provisioning and cleaning networks. Spawning instances by non-admin users in these networks and getting access to the Bare Metal services control plane is a security risk. For this reason, the provisioning and cleaning networks should be configured as non-shared networks in the `admin` tenant.

Note: When using the `flat` network interface, bare metal instances are normally spawned onto the provisioning network. This is not supported with the `neutron` interface and the deployment will fail. Please ensure a different network is chosen in the Networking service when a bare metal instance is booted from the Compute service.

Note: The provisioning and cleaning networks may be the same network or distinct networks. To ensure that communication between the Bare Metal service and the deploy ramdisk works, it is important to ensure that security groups are disabled for these networks, *or* that the default security groups allow:

- DHCP
- TFTP
- egress port used for the Bare Metal service (6385 by default)

- ingress port used for ironic-python-agent (9999 by default)
- if using *Direct deploy*, the egress port used for the Object Storage service or the local HTTP server (typically 80 or 443)
- if using iPXE, the egress port used for the HTTP server running on the ironic-conductor nodes (typically 80).

4. This step is optional and applicable only if you want to use security groups during provisioning and/or cleaning of the nodes. If not specified, default security groups are used.

   1. Define security groups in the Networking service, to be used for provisioning and/or cleaning networks.
   2. Add the list of these security group UUIDs under the `neutron` section of ironic-conductors configuration file as shown below:

      ```
      [neutron]
      ...
      cleaning_network=$CLEAN_UUID_OR_NAME
      cleaning_network_security_groups=[$LIST_OF_CLEAN_SECURITY_GROUPS]
      provisioning_network=$PROVISION_UUID_OR_NAME
      provisioning_network_security_groups=[$LIST_OF_PROVISION_SECURITY_
      ↪GROUPS]
      ```

      Multiple security groups may be applied to a given network, hence, they are specified as a list. The same security group(s) could be used for both provisioning and cleaning networks.

   Warning: If security groups are configured as described above, do not set the port_security_enabled flag to False for the corresponding Networking services network or port. This will cause the deploy to fail.

   For example: if `provisioning_network_security_groups` configuration option is used, ensure that port_security_enabled flag for the provisioning network is set to True. This flag is set to True by default; make sure not to override it by manually setting it to False.

5. Install and configure a compatible ML2 mechanism driver which supports bare metal provisioning for your switch. See ML2 plugin configuration manual for details.
6. Restart the ironic-conductor and ironic-api services after the modifications:

   - Fedora/RHEL8/CentOS8:

     ```
     sudo systemctl restart openstack-ironic-api
     sudo systemctl restart openstack-ironic-conductor
     ```

   - Ubuntu:

     ```
     sudo service ironic-api restart
     sudo service ironic-conductor restart
     ```

7. Make sure that the ironic-conductor is reachable over the provisioning network by trying to download a file from a TFTP server on it, from some non-control-plane server in that network:

```
tftp $TFTP_IP -c get $FILENAME
```

where FILENAME is the file located at the TFTP server.

See *Multi-tenancy in the Bare Metal service* for required node configuration.

Add images to the Image service

1. Build or download the user images as described in *Creating instance images*.
2. Add the user images to the Image service

   Load all the images created in the below steps into the Image service, and note the image UUIDs in the Image service for each one as it is generated.

   - For *whole disk images* just upload the image:

   ```
   $ openstack image create my-whole-disk-image --public \
     --disk-format qcow2 --container-format bare \
     --file my-whole-disk-image.qcow2
   ```

   Warning: The kernel/ramdisk pair must not be set for whole disk images, otherwise theyll be mistaken for partition images.

   - For *partition images* to be used only with *local boot* (the default) the `img_type` property must be set:

   ```
   $ openstack image create my-image --public \
     --disk-format qcow2 --container-format bare \
     --property img_type=partition --file my-image.qcow2
   ```

   - For *partition images* to be used with both *local* and *network* boot:

     Add the kernel and ramdisk images to the Image service:

     ```
     $ openstack image create my-kernel --public \
       --disk-format aki --container-format aki --file my-image.vmlinuz
     ```

     Store the image uuid obtained from the above step as `MY_VMLINUZ_UUID`.

     ```
     $ openstack image create my-image.initrd --public \
       --disk-format ari --container-format ari --file my-image.initrd
     ```

     Store the image UUID obtained from the above step as `MY_INITRD_UUID`.

     Add the *my-image* to the Image service which is going to be the OS that the user is going to run. Also associate the above created images with this OS image. These two operations can be done by executing the following command:

     ```
     $ openstack image create my-image --public \
       --disk-format qcow2 --container-format bare --property \
     ```

(continues on next page)

(continued from previous page)

```
  kernel_id=$MY_VMLINUZ_UUID --property \
  ramdisk_id=$MY_INITRD_UUID --file my-image.qcow2
```

3. Build or download the deploy images

   The deploy images are used initially for preparing the server (creating disk partitions) before the actual OS can be deployed.

   There are several methods to build or download deploy images, please read the *Building or downloading a deploy ramdisk image* section.

4. Add the deploy images to the Image service

   Add the deployment kernel and ramdisk images to the Image service:

```
$ openstack image create deploy-vmlinuz --public \
  --disk-format aki --container-format aki \
  --file ironic-python-agent.vmlinuz
```

   Store the image UUID obtained from the above step as `DEPLOY_VMLINUZ_UUID`.

```
$ openstack image create deploy-initrd --public \
  --disk-format ari --container-format ari \
  --file ironic-python-agent.initramfs
```

   Store the image UUID obtained from the above step as `DEPLOY_INITRD_UUID`.

Create flavors for use with the Bare Metal service

Youll need to create a special bare metal flavor in the Compute service. The flavor is mapped to the bare metal node through the nodes `resource_class` field (available starting with Bare Metal API version 1.21). A flavor can request *exactly one* instance of a bare metal resource class.

Note that when creating the flavor, its useful to add the `RAM_MB` and `CPU` properties as a convenience to users, although they are not used for scheduling. The `DISK_GB` property is also not used for scheduling, but is still used to determine the root partition size.

1. Change these to match your hardware:

```
$ RAM_MB=1024
$ CPU=2
$ DISK_GB=100
```

2. Create the bare metal flavor by executing the following command:

```
$ openstack flavor create --ram $RAM_MB --vcpus $CPU --disk $DISK_GB \
  my-baremetal-flavor
```

Note: You can add `--id <id>` to specify an ID for the flavor.

See the docs on this command for other options that may be specified.

After creation, associate each flavor with one custom resource class. The name of a custom resource class that corresponds to a nodes resource class (in the Bare Metal service) is:

- the bare metal nodes resource class all upper-cased
- prefixed with `CUSTOM_`
- all punctuation replaced with an underscore

For example, if the resource class is named `baremetal-small`, associate the flavor with this custom resource class via:

```
$ openstack flavor set --property resources:CUSTOM_BAREMETAL_SMALL=1 my-
↪baremetal-flavor
```

Another set of flavor properties must be used to disable scheduling based on standard properties for a bare metal flavor:

```
$ openstack flavor set --property resources:VCPU=0 my-baremetal-flavor
$ openstack flavor set --property resources:MEMORY_MB=0 my-baremetal-flavor
$ openstack flavor set --property resources:DISK_GB=0 my-baremetal-flavor
```

Example

If you want to define a class of nodes called `baremetal.with-GPU`, start with tagging some nodes with it:

```
$ baremetal node set <node> --resource-class baremetal.with-GPU
```

> **Warning:** It is possible to **add** a resource class to `active` nodes, but it is not possible to **replace** an existing resource class on them.

Then you can update your flavor to request the resource class instead of the standard properties:

```
$ openstack flavor set --property resources:CUSTOM_BAREMETAL_WITH_GPU=1 my-
↪baremetal-flavor
$ openstack flavor set --property resources:VCPU=0 my-baremetal-flavor
$ openstack flavor set --property resources:MEMORY_MB=0 my-baremetal-flavor
$ openstack flavor set --property resources:DISK_GB=0 my-baremetal-flavor
```

Note how `baremetal.with-GPU` in the nodes `resource_class` field becomes `CUSTOM_BAREMETAL_WITH_GPU` in the flavors properties.

Scheduling based on traits

Starting with the Queens release, the Compute service supports scheduling based on qualitative attributes using traits. Starting with Bare Metal REST API version 1.37, it is possible to assign a list of traits to each bare metal node. Traits assigned to a bare metal node will be assigned to the corresponding resource provider in the Compute service placement API.

When creating a flavor in the Compute service, required traits may be specified via flavor properties. The Compute service will then schedule instances only to bare metal nodes with all of the required traits.

Traits can be either standard or custom. Standard traits are listed in the os_traits library. Custom traits must meet the following requirements:

- prefixed with `CUSTOM_`
- contain only upper case characters A to Z, digits 0 to 9, or underscores
- no longer than 255 characters in length

A bare metal node can have a maximum of 50 traits.

Example

To add the standard trait `HW_CPU_X86_VMX` and a custom trait `CUSTOM_TRAIT1` to a node:

```
$ baremetal node add trait <node> CUSTOM_TRAIT1 HW_CPU_X86_VMX
```

Then, update the flavor to require these traits:

```
$ openstack flavor set --property trait:CUSTOM_TRAIT1=required my-baremetal-
↪flavor
$ openstack flavor set --property trait:HW_CPU_X86_VMX=required my-baremetal-
↪flavor
```

2.1.6 Set up the drivers for the Bare Metal service

Enabling drivers and hardware types

Introduction

The Bare Metal service delegates actual hardware management to **drivers**. *Drivers*, also called *hardware types*, consist of *hardware interfaces*: sets of functionality dealing with some aspect of bare metal provisioning in a vendor-specific way. There are generic **hardware types** (eg. `redfish` and `ipmi`), and vendor-specific ones (eg. `ilo` and `irmc`).

Note: Starting with the Rocky release, the terminologies *driver*, *dynamic driver*, and *hardware type* have the same meaning in the scope of Bare Metal service.

Enabling hardware types

Hardware types are enabled in the configuration file of the **ironic-conductor** service by setting the `enabled_hardware_types` configuration option, for example:

```
[DEFAULT]
enabled_hardware_types = ipmi,redfish
```

Due to the drivers dynamic nature, they also require configuring enabled hardware interfaces.

Note: All available hardware types and interfaces are listed in setup.cfg file in the source code tree.

Enabling hardware interfaces

There are several types of hardware interfaces:

bios manages configuration of the BIOS settings of a bare metal node. This interface is vendor-specific and can be enabled via the `enabled_bios_interfaces` option:

```
[DEFAULT]
enabled_hardware_types = <hardware_type_name>
enabled_bios_interfaces = <bios_interface_name>
```

See *BIOS Configuration* for details.

boot manages booting of both the deploy ramdisk and the user instances on the bare metal node. See *Boot interfaces* for details.

Boot interface implementations are often vendor specific, and can be enabled via the `enabled_boot_interfaces` option:

```
[DEFAULT]
enabled_hardware_types = ipmi,ilo
enabled_boot_interfaces = pxe,ilo-virtual-media
```

Boot interfaces with `pxe` in their name require *Configuring PXE and iPXE*. There are also a few hardware-specific boot interfaces - see *Drivers, Hardware Types and Hardware Interfaces* for their required configuration.

console manages access to the serial console of a bare metal node. See *Configuring Web or Serial Console* for details.

deploy defines how the image gets transferred to the target disk. See *Deploy Interfaces* for an explanation of the difference between supported deploy interfaces.

The deploy interfaces can be enabled as follows:

```
[DEFAULT]
enabled_hardware_types = ipmi,redfish
enabled_deploy_interfaces = direct,ramdisk
```

Note: The `direct` deploy interface requires the Object Storage service or an HTTP service

inspect implements fetching hardware information from nodes. Can be implemented out-of-band (via contacting the nodes BMC) or in-band (via booting a ramdisk on a node). The latter implementation is called `inspector` and uses a separate service called ironic-inspector. Example:

```
[DEFAULT]
enabled_hardware_types = ipmi,ilo,irmc
enabled_inspect_interfaces = ilo,irmc,inspector
```

See *Hardware Inspection* for more details.

management provides additional hardware management actions, like getting or setting boot devices. This interface is usually vendor-specific, and its name often matches the name of the hardware type (with `ipmitool` being a notable exception). For example:

```
[DEFAULT]
enabled_hardware_types = ipmi,redfish,ilo,irmc
enabled_management_interfaces = ipmitool,redfish,ilo,irmc
```

Using `ipmitool` requires *Configuring IPMI support*. See *Drivers, Hardware Types and Hardware Interfaces* for the required configuration of each driver.

network connects/disconnects bare metal nodes to/from virtual networks. See *Configure tenant networks* for more details.

power runs power actions on nodes. Similar to the management interface, it is usually vendor-specific, and its name often matches the name of the hardware type (with `ipmitool` being again an exception). For example:

```
[DEFAULT]
enabled_hardware_types = ipmi,redfish,ilo,irmc
enabled_power_interfaces = ipmitool,redfish,ilo,irmc
```

Using `ipmitool` requires *Configuring IPMI support*. See *Drivers, Hardware Types and Hardware Interfaces* for the required configuration of each driver.

raid manages building and tearing down RAID on nodes. Similar to inspection, it can be implemented either out-of-band or in-band (via `agent` implementation). See *RAID Configuration* for details. For example:

```
[DEFAULT]
enabled_hardware_types = ipmi,redfish,ilo,irmc
enabled_raid_interfaces = agent,no-raid
```

storage manages the interaction with a remote storage subsystem, such as the Block Storage service, and helps facilitate booting from a remote volume. This interface ensures that volume target and connector information is updated during the lifetime of a deployed instance. See *Boot From Volume* for more details.

This interface defaults to a `noop` driver as it is considered an opt-in interface which requires additional configuration by the operator to be usable.

For example:

```
[DEFAULT]
enabled_hardware_types = ipmi,irmc
enabled_storage_interfaces = cinder,noop
```

vendor is a place for vendor extensions to be exposed in API. See *Vendor Methods* for details.

```
[DEFAULT]
enabled_hardware_types = ipmi,redfish,ilo,irmc
enabled_vendor_interfaces = ipmitool,no-vendor
```

Here is a complete configuration example, enabling two generic protocols, IPMI and Redfish, with a few additional features:

```
[DEFAULT]
enabled_hardware_types = ipmi,redfish
enabled_boot_interfaces = pxe
enabled_console_interfaces = ipmitool-socat,no-console
enabled_deploy_interfaces = direct
enabled_inspect_interfaces = inspector
enabled_management_interfaces = ipmitool,redfish
enabled_network_interfaces = flat,neutron
enabled_power_interfaces = ipmitool,redfish
enabled_raid_interfaces = agent
enabled_storage_interfaces = cinder,noop
enabled_vendor_interfaces = ipmitool,no-vendor
```

Note that some interfaces have implementations named `no-<TYPE>` where <TYPE> is the interface type. These implementations do nothing and return errors when used from API.

Hardware interfaces in multi-conductor environments

When enabling hardware types and their interfaces, make sure that for every enabled hardware type, the whole set of enabled interfaces matches for all conductors. However, different conductors can have different hardware types enabled.

For example, you can have two conductors with the following configuration respectively:

```
[DEFAULT]
enabled_hardware_types = ipmi
enabled_deploy_interfaces = direct
enabled_power_interfaces = ipmitool
enabled_management_interfaces = ipmitool
```

```
[DEFAULT]
enabled_hardware_types = redfish
enabled_deploy_interfaces = ansible
enabled_power_interfaces = redfish
enabled_management_interfaces = redfish
```

But you cannot have two conductors with the following configuration respectively:

```
[DEFAULT]
enabled_hardware_types = ipmi,redfish
enabled_deploy_interfaces = direct
enabled_power_interfaces = ipmitool,redfish
enabled_management_interfaces = ipmitool,redfish
```

```
[DEFAULT]
enabled_hardware_types = redfish
enabled_deploy_interfaces = ansible
enabled_power_interfaces = redfish
enabled_management_interfaces = redfish
```

This is because the `redfish` hardware type will have different enabled *deploy* interfaces on these conductors. It would have been fine, if the second conductor had `enabled_deploy_interfaces = direct` instead of `ansible`.

This situation is not detected by the Bare Metal service, but it can cause inconsistent behavior in the API, when node functionality will depend on which conductor it gets assigned to.

Note: We dont treat this as an error, because such *temporary* inconsistency is inevitable during a rolling upgrade or a configuration update.

Configuring interface defaults

When an operator does not provide an explicit value for one of the interfaces (when creating a node or updating its driver), the default value is calculated as described in *Defaults for hardware interfaces*. It is also possible to override the defaults for any interfaces by setting one of the options named `default_<IFACE>_interface`, where `<IFACE>` is the interface name. For example:

```
[DEFAULT]
default_deploy_interface = direct
default_network_interface = neutron
```

This configuration forces the default *deploy* interface to be `direct` and the default *network* interface to be `neutron` for all hardware types.

The defaults are calculated and set on a node when creating it or updating its hardware type. Thus, changing these configuration options has no effect on existing nodes.

Warning: The default interface implementation must be configured the same way across all conductors in the cloud, except maybe for a short period of time during an upgrade or configuration update. Otherwise the default implementation will depend on which conductor handles which node, and this mapping is not predictable or even persistent.

Warning: These options should be used with care. If a hardware type does not support the provided default implementation, its users will have to always provide an explicit value for this interface when creating a node.

Configuring PXE and iPXE

DHCP server setup

A DHCP server is required by PXE/iPXE client. You need to follow steps below.

1. Set the `[dhcp]/dhcp_provider` to `neutron` in the Bare Metal Services configuration file (`/etc/ironic/ironic.conf`):

   Note: Refer *Configure tenant networks* for details. The `dhcp_provider` configuration is already set by the configuration defaults, and when you create subnet, DHCP is also enabled if you do not add any dhcp options at openstack subnet create command.

2. Enable DHCP in the subnet of PXE network.
3. Set the ip address range in the subnet for DHCP.

   Note: Refer *Configure the Networking service for bare metal provisioning* for details about the two precedent steps.

4. Connect the openstack DHCP agent to the external network through the OVS bridges and the interface `eth2`.

   Note: Refer *Configure the Networking service for bare metal provisioning* for details. You do not require this part if br-int, br-eth2 and eth2 are already connected.

5. Configure the host ip at `br-eth2`. If it locates at `eth2`, do below:

```
ip addr del 192.168.2.10/24 dev eth2
ip addr add 192.168.2.10/24 dev br-eth2
```

   Note: Replace eth2 with the interface on the network node which you are using to connect to the Bare Metal service.

TFTP server setup

In order to deploy instances via PXE, a TFTP server needs to be set up on the Bare Metal service nodes which run the `ironic-conductor`.

1. Make sure the tftp root directory exist and can be written to by the user the `ironic-conductor` is running as. For example:

```
sudo mkdir -p /tftpboot
sudo chown -R ironic /tftpboot
```

2. Install tftp server:

   Ubuntu:

```
sudo apt-get install xinetd tftpd-hpa
```

   RHEL8/CentOS8/Fedora:

```
sudo dnf install tftp-server xinetd
```

   SUSE:

```
sudo zypper install tftp xinetd
```

3. Using xinetd to provide a tftp server setup to serve `/tftpboot`. Create or edit `/etc/xinetd.d/tftp` as below:

```
service tftp
{
  protocol        = udp
  port            = 69
  socket_type     = dgram
  wait            = yes
  user            = root
  server          = /usr/sbin/in.tftpd
  server_args     = -v -v -v -v -v --map-file /tftpboot/map-file /tftpboot
  disable         = no
  # This is a workaround for Fedora, where TFTP will listen only on
  # IPv6 endpoint, if IPv4 flag is not used.
  flags           = IPv4
}
```

   and restart the `xinetd` service:

   Ubuntu:

```
sudo service xinetd restart
```

   Fedora/RHEL8/CentOS8/SUSE:

```
sudo systemctl restart xinetd
```

Note: In certain environments the networks MTU may cause TFTP UDP packets to get fragmented. Certain PXE firmwares struggle to reconstruct the fragmented packets which can cause significant slow down or even prevent the server from PXE booting. In order to avoid this, TFTPd provides an option to limit the packet size so that it they do not get fragmented. To set this additional option in the server_args above:

```
--blocksize <MAX MTU minus 32>
```

4. Create a map file in the tftp boot directory (`/tftpboot`):

```
echo 're ^(/tftpboot/) /tftpboot/\2' > /tftpboot/map-file
echo 're ^/tftpboot/ /tftpboot/' >> /tftpboot/map-file
echo 're ^(^/) /tftpboot/\1' >> /tftpboot/map-file
echo 're ^([^/]) /tftpboot/\1' >> /tftpboot/map-file
```

UEFI PXE - Grub setup

In order to deploy instances with PXE on bare metal nodes which support UEFI, perform these additional steps on the ironic conductor node to configure the PXE UEFI environment.

1. Install Grub2 and shim packages:

   Ubuntu (18.04LTS and later):

```
sudo apt-get install grub-efi-amd64-signed shim-signed
```

   RHEL8/CentOS8/Fedora:

```
sudo dnf install grub2-efi shim
```

   SUSE:

```
sudo zypper install grub2-x86_64-efi shim
```

2. Copy grub and shim boot loader images to `/tftpboot` directory:

   Ubuntu (18.04LTS and later):

```
sudo cp /usr/lib/shim/shimx64.efi.signed /tftpboot/bootx64.efi
sudo cp /usr/lib/grub/x86_64-efi-signed/grubnetx64.efi.signed /tftpboot/
↪grubx64.efi
```

   Fedora:

```
sudo cp /boot/efi/EFI/fedora/shim.efi /tftpboot/bootx64.efi
sudo cp /boot/efi/EFI/fedora/grubx64.efi /tftpboot/grubx64.efi
```

   RHEL8/CentOS8:

```
sudo cp /boot/efi/EFI/centos/shim.efi /tftpboot/bootx64.efi
sudo cp /boot/efi/EFI/centos/grubx64.efi /tftpboot/grubx64.efi
```

SUSE:

```
sudo cp /usr/lib64/efi/shim.efi /tftpboot/bootx64.efi
sudo cp /usr/lib/grub2/x86_64-efi/grub.efi /tftpboot/grubx64.efi
```

3. Update the bare metal node with `boot_mode:uefi` capability in nodes properties field. See *Boot mode support* for details.
4. Make sure that bare metal node is configured to boot in UEFI boot mode and boot device is set to network/pxe.

Note: Some drivers, e.g. `ilo`, `irmc` and `redfish`, support automatic setting of the boot mode during deployment. This step is not required for them. Please check *Drivers, Hardware Types and Hardware Interfaces* for information on whether your driver requires manual UEFI configuration.

Legacy BIOS - Syslinux setup

In order to deploy instances with PXE on bare metal using Legacy BIOS boot mode, perform these additional steps on the ironic conductor node.

1. Install the syslinux package with the PXE boot images:

   Ubuntu (16.04LTS and later):

   ```
   sudo apt-get install syslinux-common pxelinux
   ```

   RHEL8/CentOS8/Fedora:

   ```
   sudo dnf install syslinux-tftpboot
   ```

   SUSE:

   ```
   sudo zypper install syslinux
   ```

2. Copy the PXE image to `/tftpboot`. The PXE image might be found at[1]:

   Ubuntu (16.04LTS and later):

   ```
   sudo cp /usr/lib/PXELINUX/pxelinux.0 /tftpboot
   ```

   RHEL8/CentOS8/SUSE:

   ```
   sudo cp /usr/share/syslinux/pxelinux.0 /tftpboot
   ```

3. If whole disk images need to be deployed via PXE-netboot, copy the chain.c32 image to `/tftpboot` to support it:

   Ubuntu (16.04LTS and later):

[1] On **Fedora/RHEL** the `syslinux-tftpboot` package already installs the library modules and PXE image at `/tftpboot`. If the TFTP server is configured to listen to a different directory you should copy the contents of `/tftpboot` to the configured directory

```
sudo cp /usr/lib/syslinux/modules/bios/chain.c32 /tftpboot
```

Fedora:

```
sudo cp /boot/extlinux/chain.c32 /tftpboot
```

RHEL8/CentOS8/SUSE:

```
sudo cp /usr/share/syslinux/chain.c32 /tftpboot/
```

4. If the version of syslinux is **greater than** 4 we also need to make sure that we copy the library modules into the `/tftpboot` directory[2]?. For example, for Ubuntu run:

```
sudo cp /usr/lib/syslinux/modules/*/ldlinux.* /tftpboot
```

5. Update the bare metal node with `boot_mode:bios` capability in nodes properties field. See *Boot mode support* for details.
6. Make sure that bare metal node is configured to boot in Legacy BIOS boot mode and boot device is set to network/pxe.

iPXE setup

If you will be using iPXE to boot instead of PXE, iPXE needs to be set up on the Bare Metal service node(s) where `ironic-conductor` is running.

1. Make sure these directories exist and can be written to by the user the `ironic-conductor` is running as. For example:

```
sudo mkdir -p /tftpboot
sudo mkdir -p /httpboot
sudo chown -R ironic /tftpboot
sudo chown -R ironic /httpboot
```

2. Create a map file in the tftp boot directory (`/tftpboot`):

```
echo 'r ^([^/]) /tftpboot/\1' > /tftpboot/map-file
echo 'r ^(/tftpboot/) /tftpboot/\2' >> /tftpboot/map-file
```

3. Set up TFTP and HTTP servers.

   These servers should be running and configured to use the local /tftpboot and /httpboot directories respectively, as their root directories. (Setting up these servers is outside the scope of this install guide.)

   These root directories need to be mounted locally to the `ironic-conductor` services, so that the services can access them.

   The Bare Metal services configuration file (/etc/ironic/ironic.conf) should be edited accordingly to specify the TFTP and HTTP root directories and server addresses. For example:

[2] http://www.syslinux.org/wiki/index.php/Library_modules

```
[pxe]

# Ironic compute node's tftp root path. (string value)
tftp_root=/tftpboot

# IP address of Ironic compute node's tftp server. (string
# value)
tftp_server=192.168.0.2

[deploy]
# Ironic compute node's http root path. (string value)
http_root=/httpboot

# Ironic compute node's HTTP server URL. Example:
# http://192.1.2.3:8080 (string value)
http_url=http://192.168.0.2:8080
```

See also: *Deploying outside of the provisioning network*.

4. Install the iPXE package with the boot images:

   Ubuntu:

   ```
   apt-get install ipxe
   ```

   RHEL8/CentOS8/Fedora:

   ```
   dnf install ipxe-bootimgs
   ```

   Note: SUSE does not provide a package containing iPXE boot images. If you are using SUSE or if the packaged version of the iPXE boot image doesnt work, you can download a prebuilt one from http://boot.ipxe.org or build one image from source, see http://ipxe.org/download for more information.

5. Copy the iPXE boot image (`undionly.kpxe` for **BIOS** and `ipxe.efi` for **UEFI**) to `/tftpboot`. The binary might be found at:

   Ubuntu:

   ```
   cp /usr/lib/ipxe/{undionly.kpxe,ipxe.efi,snponly.efi} /tftpboot
   ```

   Fedora/RHEL8/CentOS8:

   ```
   cp /usr/share/ipxe/{undionly.kpxe,ipxe-x86_64.efi,ipxe-snponly-x86_64.efi}
   ↪ /tftpboot
   ```

   Note: `snponly` variants may not be available for all distributions.

6. Enable/Configure iPXE overrides in the Bare Metal Services configuration file **if required** (/etc/ironic/ironic.conf):

```
[pxe]

# Neutron bootfile DHCP parameter. (string value)
ipxe_bootfile_name=undionly.kpxe

# Bootfile DHCP parameter for UEFI boot mode. (string value)
uefi_ipxe_bootfile_name=ipxe.efi

# Template file for PXE configuration. (string value)
ipxe_config_template=$pybasedir/drivers/modules/ipxe_config.template
```

Note: Most UEFI systems have integrated networking which means the `[pxe]uefi_ipxe_bootfile_name` setting should be set to `snponly.efi` or `ipxe-snponly-x86_64.efi` if its available for your distribution.

Note: Setting the iPXE parameters noted in the code block above to no value, in other words setting a line to something like `ipxe_bootfile_name=` will result in ironic falling back to the default values of the non-iPXE PXE settings. This is for backwards compatability.

7. Ensure iPXE is the default PXE, if applicable.

   In earlier versions of ironic, a `[pxe]ipxe_enabled` setting allowing operators to declare the behavior of the conductor to exclusively operate as if only iPXE was to be used. As time moved on, iPXE functionality was moved to its own `ipxe` boot interface.

   If you want to emulate that same behavior, set the following in the configuration file (/etc/ironic/ironic.conf):

   ```
   [DEFAULT]
   default_boot_interface=ipxe
   enabled_boot_interfaces=ipxe,pxe
   ```

   Note: The `[DEFAULT]enabled_boot_interfaces` setting may be exclusively set to `ipxe`, however ironic has multiple interfaces available depending on the hardware types available for use.

8. It is possible to configure the Bare Metal service in such a way that nodes will boot into the deploy image directly from Object Storage. Doing this avoids having to cache the images on the ironic-conductor host and serving them via the ironic-conductors *HTTP server*. This can be done if:

   1. the Image Service is used for image storage;
   2. the images in the Image Service are internally stored in Object Storage;
   3. the Object Storage supports generating temporary URLs for accessing objects stored in it. Both the OpenStack Swift and RADOS Gateway provide support for this.
      - See *Ceph Object Gateway support* on how to configure the Bare Metal Service with RADOS Gateway as the Object Storage.

   Configure this by setting the `[pxe]/ipxe_use_swift` configuration option to `True` as follows:

```
[pxe]

# Download deploy images directly from swift using temporary
# URLs. If set to false (default), images are downloaded to
# the ironic-conductor node and served over its local HTTP
# server. Applicable only when 'ipxe_enabled' option is set to
# true. (boolean value)
ipxe_use_swift=True
```

Although the *HTTP server* still has to be deployed and configured (as it will serve iPXE boot script and boot configuration files for nodes), such configuration will shift some load from ironic-conductor hosts to the Object Storage service which can be scaled horizontally.

Note that when SSL is enabled on the Object Storage service you have to ensure that iPXE firmware on the nodes can indeed boot from generated temporary URLs that use HTTPS protocol.

9. Restart the `ironic-conductor` process:

   Fedora/RHEL8/CentOS8/SUSE:

   ```
   sudo systemctl restart openstack-ironic-conductor
   ```

   Ubuntu:

   ```
   sudo service ironic-conductor restart
   ```

PXE multi-architecture setup

It is possible to deploy servers of different architecture by one conductor. To use this feature, architecture-specific boot and template files must be configured using the configuration options `[pxe]pxe_bootfile_name_by_arch` and `[pxe]pxe_config_template_by_arch` respectively, in the Bare Metal services configuration file (/etc/ironic/ironic.conf).

These two options are dictionary values; the key is the architecture and the value is the boot (or config template) file. A nodes `cpu_arch` property is used as the key to get the appropriate boot file and template file. If the nodes `cpu_arch` is not in the dictionary, the configuration options (in [pxe] group) `pxe_bootfile_name`, `pxe_config_template`, `uefi_pxe_bootfile_name` and `uefi_pxe_config_template` will be used instead.

In the following example, since x86 and x86_64 keys are not in the `pxe_bootfile_name_by_arch` or `pxe_config_template_by_arch` options, x86 and x86_64 nodes will be deployed by pxelinux.0 or bootx64.efi, depending on the nodes `boot_mode` capability (bios or uefi). However, aarch64 nodes will be deployed by grubaa64.efi, and ppc64 nodes by bootppc64:

```
[pxe]

# Bootfile DHCP parameter. (string value)
pxe_bootfile_name=pxelinux.0

# On ironic-conductor node, template file for PXE
# configuration. (string value)
pxe_config_template = $pybasedir/drivers/modules/pxe_config.template
```

(continues on next page)

(continued from previous page)

```
# Bootfile DHCP parameter for UEFI boot mode. (string value)
uefi_pxe_bootfile_name=bootx64.efi

# On ironic-conductor node, template file for PXE
# configuration for UEFI boot loader. (string value)
uefi_pxe_config_template=$pybasedir/drivers/modules/pxe_grub_config.template

# Bootfile DHCP parameter per node architecture. (dict value)
pxe_bootfile_name_by_arch=aarch64:grubaa64.efi,ppc64:bootppc64

# On ironic-conductor node, template file for PXE
# configuration per node architecture. For example:
# aarch64:/opt/share/grubaa64_pxe_config.template (dict value)
pxe_config_template_by_arch=aarch64:pxe_grubaa64_config.template,ppc64:pxe_
↪ppc64_config.template
```

Note: The grub implementation may vary on different architecture, you may need to tweak the pxe config template for a specific arch. For example, grubaa64.efi shipped with CentoOS7 does not support `linuxefi` and `initrdefi` commands, youll need to switch to use `linux` and `initrd` command instead.

Note: A `[pxe]ipxe_bootfile_name_by_arch` setting is available for multi-arch iPXE based deployment, and defaults to the same behavior as the comperable `[pxe]pxe_bootfile_by_arch` setting for standard PXE.

PXE timeouts tuning

Because of its reliance on UDP-based protocols (DHCP and TFTP), PXE is particularly vulnerable to random failures during the booting stage. If the deployment ramdisk never calls back to the bare metal conductor, the build will be aborted, and the node will be moved to the `deploy failed` state, after the deploy callback timeout. This timeout can be changed via the *conductor.deploy_callback_timeout* configuration option.

Starting with the Train release, the Bare Metal service can retry PXE boot if it takes too long. The timeout is defined via *pxe.boot_retry_timeout* and must be smaller than the `deploy_callback_timeout`, otherwise it will have no effect.

For example, the following configuration sets the overall timeout to 60 minutes, allowing two retries after 20 minutes:

```
[conductor]
deploy_callback_timeout = 3600

[pxe]
boot_retry_timeout = 1200
```

PXE artifacts

Ironic features the capability to load PXE artifacts into the conductor startup, minimizing the need for external installation and configuration management tooling from having to do additional work to facilitate.

While this is an advanced feature, and destination file names must match existing bootloader configured filenames.

For example, if using iPXE and GRUB across interfaces, you may desire a configuration similar to this example.

```
[pxe]
loader_file_paths = ipxe.efi:/usr/share/ipxe/ipxe-snponly-x86_64.efi,undionly.
↪kpxe:/usr/share/ipxe/undionly.kpxe,bootx64.efi,/boot/efi/EFI/boot/grubx64.
↪efi,bootx64.efi:/boot/efi/EFI/boot/BOOTX64.EFI
```

If you choose to use relative paths as part of your destination, those paths will be created using configuration parameter `[pxe]dir_permission` where as actual files copied are set with the configuration parameter `[pxe]file_permission`. Absolute destination paths are not supported and will result in ironic failing to start up as it is a misconfiguration of the deployment.

Configuring IPMI support

Installing ipmitool command

To enable one of the drivers that use IPMI protocol for power and management actions (for example, `ipmi`), the `ipmitool` command must be present on the service node(s) where `ironic-conductor` is running. On most distros, it is provided as part of the `ipmitool` package. Source code is available at http://ipmitool.sourceforge.net/.

Warning: Certain distros, notably Mac OS X and SLES, install `openipmi` instead of `ipmitool` by default. This driver is not compatible with `openipmi` as it relies on error handling options not provided by this tool.

Please refer to the *IPMI driver* for information on how to configure and use IPMItool-based drivers.

Configuring hardware

IPMI is a relatively old protocol and may require additional set up on the hardware side that the Bare Metal service cannot do automatically:

1. Make sure IPMI is enabled and the account you use have the permissions to change power and boot devices. By default the adminstrator rights are expected, you can change it: see *Using a different privilege level*.
2. Make sure the cipher suites are configured for maximum security. Suite 17 is recommended, 3 can be used if its not available. Cipher suite 0 **must** be disabled as it provides unauthenticated access to the BMC.

See also:

Cipher suites

3. Make sure the boot mode correspond to the expected boot mode on the node (see *Boot mode support*). Some hardware is able to change the boot mode to the requested by Ironic, some does not.

Validation and troubleshooting

Check that you can connect to, and authenticate with, the IPMI controller in your bare metal server by running `ipmitool`:

```
ipmitool -I lanplus -H <ip-address> -U <username> -P <password> chassis power␣
↪status
```

where `<ip-address>` is the IP of the IPMI controller you want to access. This is not the bare metal nodes main IP. The IPMI controller should have its own unique IP.

If the above command doesnt return the power status of the bare metal server, check that

- `ipmitool` is installed and is available via the `$PATH` environment variable.
- The IPMI controller on your bare metal server is turned on.
- The IPMI controller credentials and IP address passed in the command are correct.
- The conductor node has a route to the IPMI controller. This can be checked by just pinging the IPMI controller IP from the conductor node.

IPMI configuration

If there are slow or unresponsive BMCs in the environment, the `min_command_interval` configuration option in the `[ipmi]` section may need to be raised. The default is fairly conservative, as setting this timeout too low can cause older BMCs to crash and require a hard-reset.

Collecting sensor data

Bare Metal service supports sending IPMI sensor data to Telemetry with certain hardware types, such as `ipmi`, `ilo` and `irmc`. By default, support for sending IPMI sensor data to Telemetry is disabled. If you want to enable it, you should make the following two changes in `ironic.conf`:

```
[conductor]
send_sensor_data = true
[oslo_messaging_notifications]
driver = messagingv2
```

If you want to customize the sensor types which will be sent to Telemetry, change the `send_sensor_data_types` option. For example, the below settings will send information about temperature, fan, voltage from sensors to the Telemetry service:

```
send_sensor_data_types=Temperature,Fan,Voltage
```

Supported sensor types are defined by the Telemetry service, currently these are `Temperature`, `Fan`, `Voltage`, `Current`. Special value `All` (the default) designates all supported sensor types.

2.1.7 Enrollment

After all the services have been properly configured, you should enroll your hardware with the Bare Metal service, and confirm that the Compute service sees the available hardware. The nodes will be visible to the Compute service once they are in the `available` provision state.

Note: After enrolling nodes with the Bare Metal service, the Compute service will not be immediately notified of the new resources. The Compute services resource tracker syncs periodically, and so any changes made directly to the Bare Metal services resources will become visible in the Compute service only after the next run of that periodic task. More information is in the *Troubleshooting* section.

Note: Any bare metal node that is visible to the Compute service may have a workload scheduled to it, if both the `power` and `management` interfaces pass the `validate` check. If you wish to exclude a node from the Compute services scheduler, for instance so that you can perform maintenance on it, you can set the node to maintenance mode. For more information see the *Maintenance mode* section.

Choosing a driver

When enrolling a node, the most important information to supply is *driver*. See *Enabling drivers and hardware types* for a detailed explanation of bare metal drivers, hardware types and interfaces. The `driver list` command can be used to list all drivers enabled on all hosts:

```
baremetal driver list
+---------------------+------------------------+
| Supported driver(s) | Active host(s)         |
+---------------------+------------------------+
| ipmi                | localhost.localdomain  |
+---------------------+------------------------+
```

The specific driver to use should be picked based on actual hardware capabilities and expected features. See *Drivers, Hardware Types and Hardware Interfaces* for more hints on that.

Each driver has a list of *driver properties* that need to be specified via the nodes `driver_info` field, in order for the driver to operate on node. This list consists of the properties of the hardware interfaces that the driver uses. These driver properties are available with the `driver property list` command:

```
$ baremetal driver property list ipmi
+----------------------+-------------------------------------------------------
↪-----------------------------------------------------+
| Property             | Description                                           ␣
↪                                                    |
+----------------------+-------------------------------------------------------
↪-----------------------------------------------------+
| ipmi_address         | IP address or hostname of the node. Required.         ␣
↪                                                    |
```

(continues on next page)

(continued from previous page)

```
| ipmi_password        | password. Optional.                                          ␣
↪                                                          |
| ipmi_username        | username; default is NULL user. Optional.                    ␣
↪                                                          |
| ...                  | ...                                                          ␣
↪                                                          |
| deploy_kernel        | UUID (from Glance) of the deployment kernel.␣
↪Required.                                                         |
| deploy_ramdisk       | UUID (from Glance) of the ramdisk that is mounted at␣
↪boot time. Required.                                      |
+----------------------+------------------------------------------------------------
↪------------------------------------------------------+
```

The properties marked as required must be supplied either during node creation or shortly after. Some properties may only be required for certain features.

Note on API versions

Starting with API version 1.11, the Bare Metal service added a new initial provision state of `enroll` to its state machine. When this or later API version is used, new nodes get this state instead of `available`.

Existing automation tooling that use an API version lower than 1.11 are not affected, since the initial provision state is still `available`. However, using API version 1.11 or above may break existing automation tooling with respect to node creation.

The default API version used by (the most recent) python-ironicclient is 1.9, but it may change in the future and should not be relied on.

In the examples below we will use version 1.11 of the Bare metal API. This gives us the following advantages:

- Explicit power credentials validation before leaving the `enroll` state.
- Running node cleaning before entering the `available` state.
- Not exposing half-configured nodes to the scheduler.

To set the API version for all commands, you can set the environment variable `IRONIC_API_VERSION`. For the OpenStackClient baremetal plugin, set the `OS_BAREMETAL_API_VERSION` variable to the same value. For example:

```
$ export IRONIC_API_VERSION=1.11
$ export OS_BAREMETAL_API_VERSION=1.11
```

Enrollment process

Creating a node

This section describes the main steps to enroll a node and make it available for provisioning. Some steps are shown separately for illustration purposes, and may be combined if desired.

1. Create a node in the Bare Metal service with the `node create` command. At a minimum, you must specify the driver name (for example, `ipmi`).

   This command returns the node UUID along with other information about the node. The nodes provision state will be `enroll`:

```
$ export OS_BAREMETAL_API_VERSION=1.11
$ baremetal node create --driver ipmi
+--------------+--------------------------------------+
| Property     | Value                                |
+--------------+--------------------------------------+
| uuid         | dfc6189f-ad83-4261-9bda-b27258eb1987 |
| driver_info  | {}                                   |
| extra        | {}                                   |
| driver       | ipmi                                 |
| chassis_uuid |                                      |
| properties   | {}                                   |
| name         | None                                 |
+--------------+--------------------------------------+

$ baremetal node show dfc6189f-ad83-4261-9bda-b27258eb1987
+------------------------+--------------------------------------+
| Property               | Value                                |
+------------------------+--------------------------------------+
| target_power_state     | None                                 |
| extra                  | {}                                   |
| last_error             | None                                 |
| maintenance_reason     | None                                 |
| provision_state        | enroll                               |
| uuid                   | dfc6189f-ad83-4261-9bda-b27258eb1987 |
| console_enabled        | False                                |
| target_provision_state | None                                 |
| provision_updated_at   | None                                 |
| maintenance            | False                                |
| power_state            | None                                 |
| driver                 | ipmi                                 |
| properties             | {}                                   |
| instance_uuid          | None                                 |
| name                   | None                                 |
| driver_info            | {}                                   |
| ...                    | ...                                  |
+------------------------+--------------------------------------+
```

   A node may also be referred to by a logical name as well as its UUID. A name can be assigned to the node during its creation by adding the `-n` option to the `node create` command or by updating

an existing node with the `node set` command. See *Logical Names* for examples.

2. Starting with API version 1.31 (and `python-ironicclient` 1.13), you can pick which hardware interface to use with nodes that use hardware types. Each interface is represented by a node field called `<IFACE>_interface` where `<IFACE>` in the interface type, e.g. `boot`. See *Enabling drivers and hardware types* for details on hardware interfaces.

   An interface can be set either separately:

```
$ baremetal node set $NODE_UUID --deploy-interface direct --raid-
↪interface agent
```

   or set during node creation:

```
$ baremetal node create --driver ipmi \
    --deploy-interface direct \
    --raid-interface agent
```

   If no value is provided for some interfaces, *Defaults for hardware interfaces* are used instead.

3. Update the node `driver_info` with the required driver properties, so that the Bare Metal service can manage the node:

```
$ baremetal node set $NODE_UUID \
    --driver-info ipmi_username=$USER \
    --driver-info ipmi_password=$PASS \
    --driver-info ipmi_address=$ADDRESS
```

Note: If IPMI is running on a port other than 623 (the default). The port must be added to `driver_info` by specifying the `ipmi_port` value. Example:

```
$ baremetal node set $NODE_UUID --driver-info ipmi_port=$PORT_NUMBER
```

   You may also specify all `driver_info` parameters during node creation by passing the **driver-info** option multiple times:

```
$ baremetal node create --driver ipmi \
    --driver-info ipmi_username=$USER \
    --driver-info ipmi_password=$PASS \
    --driver-info ipmi_address=$ADDRESS
```

   See *Choosing a driver* above for details on driver properties.

4. Specify a deploy kernel and ramdisk compatible with the nodes driver, for example:

```
$ baremetal node set $NODE_UUID \
    --driver-info deploy_kernel=$DEPLOY_VMLINUZ_UUID \
    --driver-info deploy_ramdisk=$DEPLOY_INITRD_UUID
```

   See *Add images to the Image service* for details.

5. Optionally you can specify the provisioning and/or cleaning network UUID or name in the nodes `driver_info`. The `neutron` network interface requires both `provisioning_network` and

`cleaning_network`, while the `flat` network interface requires the `cleaning_network` to be set either in the configuration or on the nodes. For example:

```
$ baremetal node set $NODE_UUID \
    --driver-info cleaning_network=$CLEAN_UUID_OR_NAME \
    --driver-info provisioning_network=$PROVISION_UUID_OR_NAME
```

See *Configure tenant networks* for details.

6. You must also inform the Bare Metal service of the network interface cards which are part of the node by creating a port with each NICs MAC address. These MAC addresses are passed to the Networking service during instance provisioning and used to configure the network appropriately:

```
$ baremetal port create $MAC_ADDRESS --node $NODE_UUID
```

Note: When it is time to remove the node from the Bare Metal service, the command used to remove the port is `baremetal port delete <port uuid>`. When doing so, it is important to ensure that the baremetal node is not in `maintenance` as guarding logic to prevent orphaning Neutron Virtual Interfaces (VIFs) will be overridden.

Adding scheduling information

1. Assign a *resource class* to the node. A *resource class* should represent a class of hardware in your data center, that corresponds to a Compute flavor.

   For example, lets split hardware into these three groups:

   1. nodes with a lot of RAM and powerful CPU for computational tasks,
   2. nodes with powerful GPU for OpenCL computing,
   3. smaller nodes for development and testing.

   We can define three resource classes to reflect these hardware groups, named `large-cpu`, `large-gpu` and `small` respectively. Then, for each node in each of the hardware groups, well set their `resource_class` appropriately via:

```
$ baremetal node set $NODE_UUID --resource-class $CLASS_NAME
```

   The `--resource-class` argument can also be used when creating a node:

```
$ baremetal node create --driver $DRIVER --resource-class $CLASS_NAME
```

   To use resource classes for scheduling you need to update your flavors as described in *Create flavors for use with the Bare Metal service*.

   Note: This is not required for standalone deployments, only for those using the Compute service for provisioning bare metal instances.

   ---

2. Update the nodes properties to match the actual hardware of the node:

```
$ baremetal node set $NODE_UUID \
    --property cpus=$CPU_COUNT \
    --property memory_mb=$RAM_MB \
    --property local_gb=$DISK_GB
```

As above, these can also be specified at node creation by passing the **property** option to `node create` multiple times:

```
$ baremetal node create --driver ipmi \
    --driver-info ipmi_username=$USER \
    --driver-info ipmi_password=$PASS \
    --driver-info ipmi_address=$ADDRESS \
    --property cpus=$CPU_COUNT \
    --property memory_mb=$RAM_MB \
    --property local_gb=$DISK_GB
```

These values can also be discovered during *Hardware Inspection*.

> **Warning:** The value provided for the `local_gb` property must match the size of the root device youre going to deploy on. By default **ironic-python-agent** picks the smallest disk which is not smaller than 4 GiB.
>
> If you override this logic by using root device hints (see *Specifying the disk for deployment (root device hints)*), the `local_gb` value should match the size of picked target disk.

3. If you wish to perform more advanced scheduling of the instances based on hardware capabilities, you may add metadata to each node that will be exposed to the Compute scheduler (see: ComputeCapabilitiesFilter). A full explanation of this is outside of the scope of this document. It can be done through the special `capabilities` member of node properties:

```
$ baremetal node set $NODE_UUID \
    --property capabilities=key1:val1,key2:val2
```

Some capabilities can also be discovered during *Hardware Inspection*.

4. If you wish to perform advanced scheduling of instances based on qualitative attributes of bare metal nodes, you may add traits to each bare metal node that will be exposed to the Compute scheduler (see: *Scheduling based on traits* for a more in-depth discussion of traits in the Bare Metal service). For example, to add the standard trait `HW_CPU_X86_VMX` and a custom trait `CUSTOM_TRAIT1` to a node:

```
$ baremetal node add trait $NODE_UUID \
    CUSTOM_TRAIT1 HW_CPU_X86_VMX
```

Validating node information

1. To check if Bare Metal service has the minimum information necessary for a nodes driver to be functional, you may `validate` it:

```
$ baremetal node validate $NODE_UUID
+------------+--------+--------+
| Interface  | Result | Reason |
+------------+--------+--------+
| boot       | True   |        |
| console    | True   |        |
| deploy     | True   |        |
| inspect    | True   |        |
| management | True   |        |
| network    | True   |        |
| power      | True   |        |
| raid       | True   |        |
| storage    | True   |        |
+------------+--------+--------+
```

   If the node fails validation, each driver interface will return information as to why it failed:

```
$ baremetal node validate $NODE_UUID
+------------+--------+-----------------------------------------------------------
↪-----------------------------------------------------------------------------
↪----------+
| Interface  | Result | Reason                                                    ␣
↪                                                                             ␣
↪          |
+------------+--------+-----------------------------------------------------------
↪-----------------------------------------------------------------------------
↪----------+
| boot       | True   |                                                           ␣
↪                                                                             ␣
↪          |
| console    | None   | not supported                                             ␣
↪                                                                             ␣
↪          |
| deploy     | False  | Cannot validate iSCSI deploy. Some parameters␣
↪were missing in node's instance_info. Missing are: ['root_gb', 'image_
↪source']        |
| inspect    | True   |                                                           ␣
↪                                                                             ␣
↪          |
| management | False  | Missing the following IPMI credentials in node's␣
↪driver_info: ['ipmi_address'].                                               ␣
↪          |
| network    | True   |                                                           ␣
↪                                                                             ␣
↪          |
| power      | False  | Missing the following IPMI credentials in node's␣
↪driver_info: ['ipmi_address'].                                               ␣
↪          |
```

(continues on next page)

(continued from previous page)

```
| raid       | None   | not supported                                                 ␣
↪                                                                                  ␣
↪          |
| storage    | True   |                                                               ␣
↪                                                                                  ␣
↪          |
+------------+--------+---------------------------------------------------------
↪-------------------------------------------------------------------------------
↪----------+
```

When using the Compute Service with the Bare Metal service, it is safe to ignore the deploy interfaces validation error due to lack of image information. You may continue the enrollment process. This information will be set by the Compute Service just before deploying, when an instance is requested:

```
$ baremetal node validate $NODE_UUID
+------------+--------+---------------------------------------------------------
↪-------------------------------------------------------------------------------
↪---------------------------------------+
| Interface  | Result | Reason                                                        ␣
↪                                                                                  ␣
↪                                       |
+------------+--------+---------------------------------------------------------
↪-------------------------------------------------------------------------------
↪---------------------------------------+
| boot       | False  | Cannot validate image information for node␣
↪because one or more parameters are missing from its instance_info.␣
↪Missing are: ['ramdisk', 'kernel', 'image_source'] |
| console    | True   |                                                               ␣
↪                                                                                  ␣
↪                                       |
| deploy     | False  | Cannot validate image information for node␣
↪because one or more parameters are missing from its instance_info.␣
↪Missing are: ['ramdisk', 'kernel', 'image_source'] |
| inspect    | True   |                                                               ␣
↪                                                                                  ␣
↪                                       |
| management | True   |                                                               ␣
↪                                                                                  ␣
↪                                       |
| network    | True   |                                                               ␣
↪                                                                                  ␣
↪                                       |
| power      | True   |                                                               ␣
↪                                                                                  ␣
↪                                       |
| raid       | None   | not supported                                                 ␣
↪                                                                                  ␣
↪                                       |
```

(continues on next page)

(continued from previous page)

```
| storage     | True    |                                                              ␣
↪                                                                                    ␣
↪                                         |
+------------+--------+-----------------------------------------------------------
↪-----------------------------------------------------------------------------------
↪---------------------------------------+
```

Making node available for deployment

In order for nodes to be available for deploying workloads on them, nodes must be in the `available` provision state. To do this, nodes created with API version 1.11 and above must be moved from the `enroll` state to the `manageable` state and then to the `available` state. This section can be safely skipped, if API version 1.10 or earlier is used (which is the case by default).

After creating a node and before moving it from its initial provision state of `enroll`, basic power and port information needs to be configured on the node. The Bare Metal service needs this information because it verifies that it is capable of controlling the node when transitioning the node from `enroll` to `manageable` state.

To move a node from `enroll` to `manageable` provision state:

```
$ baremetal node manage $NODE_UUID
$ baremetal node show $NODE_UUID
+------------------------+-----------------------------------------------------------
↪-----------------+
| Property               | Value                                                     ␣
↪                 |
+------------------------+-----------------------------------------------------------
↪-----------------+
| ...                    | ...                                                       ␣
↪                 |
| provision_state        | manageable                                                ␣
↪                 | <- verify correct state
| uuid                   | 0eb013bb-1e4b-4f4c-94b5-2e7468242611                      ␣
↪                 |
| ...                    | ...                                                       ␣
↪                 |
+------------------------+-----------------------------------------------------------
↪-----------------+
```

Note: Since it is an asynchronous call, the response for `baremetal node manage` will not indicate whether the transition succeeded or not. You can check the status of the operation via `baremetal node show`. If it was successful, `provision_state` will be in the desired state. If it failed, there will be information in the nodes `last_error`.

When a node is moved from the `manageable` to `available` provision state, the node will go through automated cleaning if configured to do so (see *Configure the Bare Metal service for cleaning*).

To move a node from `manageable` to `available` provision state:

```
$ baremetal node provide $NODE_UUID
$ baremetal node show $NODE_UUID
+------------------------+--------------------------------------------------------
↪----------------+
| Property               | Value                                                  ␣
↪                |
+------------------------+--------------------------------------------------------
↪----------------+
| ...                    | ...                                                    ␣
↪                |
| provision_state        | available                                              ␣
↪                | < - verify correct state
| uuid                   | 0eb013bb-1e4b-4f4c-94b5-2e7468242611                   ␣
↪                |
| ...                    | ...                                                    ␣
↪                |
+------------------------+--------------------------------------------------------
↪----------------+
```

For more details on the Bare Metal services state machine, see the *Bare Metal State Machine* documentation.

Mapping nodes to Compute cells

If the Compute service is used for scheduling, and the `discover_hosts_in_cells_interval` was not set as described in *Configure the Compute service to use the Bare Metal service*, then log into any controller node and run the following command to map the new node(s) to Compute cells:

```
nova-manage cell_v2 discover_hosts
```

Logical names

A node may also be referred to by a logical name as well as its UUID. Names can be assigned either during its creation by adding the `-n` option to the `node create` command or by updating an existing node with the `node set` command.

Node names must be unique, and conform to:

- rfc952
- rfc1123
- wiki_hostname

The node is named example in the following examples:

```
$ baremetal node create --driver ipmi --name example
```

or

```
$ baremetal node set $NODE_UUID --name example
```

Once assigned a logical name, a node can then be referred to by name or UUID interchangeably:

```
$ baremetal node create --driver ipmi --name example
+--------------+--------------------------------------+
| Property     | Value                                |
+--------------+--------------------------------------+
| uuid         | 71e01002-8662-434d-aafd-f068f69bb85e |
| driver_info  | {}                                   |
| extra        | {}                                   |
| driver       | ipmi                                 |
| chassis_uuid |                                      |
| properties   | {}                                   |
| name         | example                              |
+--------------+--------------------------------------+

$ baremetal node show example
+------------------------+--------------------------------------+
| Property               | Value                                |
+------------------------+--------------------------------------+
| target_power_state     | None                                 |
| extra                  | {}                                   |
| last_error             | None                                 |
| updated_at             | 2015-04-24T16:23:46+00:00            |
| ...                    | ...                                  |
| instance_info          | {}                                   |
+------------------------+--------------------------------------+
```

Defaults for hardware interfaces

For *hardware types*, users can request one of enabled implementations when creating or updating a node as explained in *Creating a node*.

When no value is provided for a certain interface when creating a node, or changing a nodes hardware type, the default value is used. You can use the driver details command to list the current enabled and default interfaces for a hardware type (for your deployment):

```
$ baremetal driver show ipmi
+-------------------------------+----------------+
| Field                         | Value          |
+-------------------------------+----------------+
| default_boot_interface        | pxe            |
| default_console_interface     | no-console     |
| default_deploy_interface      | direct         |
| default_inspect_interface     | no-inspect     |
| default_management_interface  | ipmitool       |
| default_network_interface     | flat           |
| default_power_interface       | ipmitool       |
| default_raid_interface        | no-raid        |
| default_vendor_interface      | no-vendor      |
| enabled_boot_interfaces       | pxe            |
```

(continues on next page)

(continued from previous page)

```
| enabled_console_interfaces     | no-console     |
| enabled_deploy_interfaces      | direct         |
| enabled_inspect_interfaces     | no-inspect     |
| enabled_management_interfaces | ipmitool       |
| enabled_network_interfaces     | flat, noop     |
| enabled_power_interfaces       | ipmitool       |
| enabled_raid_interfaces        | no-raid, agent |
| enabled_vendor_interfaces      | no-vendor      |
| hosts                          | ironic-host-1  |
| name                           | ipmi           |
| type                           | dynamic        |
+--------------------------------+----------------+
```

The defaults are calculated as follows:

1. If the `default_<IFACE>_interface` configuration option (where `<IFACE>` is the interface name) is set, its value is used as the default.

   If this implementation is not compatible with the nodes hardware type, an error is returned to a user. An explicit value has to be provided for the nodes `<IFACE>_interface` field in this case.

2. Otherwise, the first supported implementation that is enabled by an operator is used as the default.

   A list of supported implementations is calculated by taking the intersection between the implementations supported by the nodes hardware type and implementations enabled by the `enabled_<IFACE>_interfaces` option (where `<IFACE>` is the interface name). The calculation preserves the order of items, as provided by the hardware type.

   If the list of supported implementations is not empty, the first one is used. Otherwise, an error is returned to a user. In this case, an explicit value has to be provided for the `<IFACE>_interface` field.

See *Enabling drivers and hardware types* for more details on configuration.

Example

Consider the following configuration (shortened for simplicity):

```
[DEFAULT]
enabled_hardware_types = ipmi,redfish
enabled_console_interfaces = no-console,ipmitool-shellinabox
enabled_deploy_interfaces = direct
enabled_management_interfaces = ipmitool,redfish
enabled_power_interfaces = ipmitool,redfish
default_deploy_interface = ansible
```

A new node is created with the `ipmi` driver and no interfaces specified:

```
$ export OS_BAREMETAL_API_VERSION=1.31
$ baremetal node create --driver ipmi
+--------------+--------------------------------------+
| Property     | Value                                |
```

(continues on next page)

(continued from previous page)

```
+--------------+--------------------------------------+
| uuid         | dfc6189f-ad83-4261-9bda-b27258eb1987 |
| driver_info  | {}                                   |
| extra        | {}                                   |
| driver       | ipmi                                 |
| chassis_uuid |                                      |
| properties   | {}                                   |
| name         | None                                 |
+--------------+--------------------------------------+
```

Then the defaults for the interfaces that will be used by the node in this example are calculated as follows:

deploy An explicit value of `ansible` is provided for `default_deploy_interface`, so it is used.

power No default is configured. The `ipmi` hardware type supports only `ipmitool` power. The intersection between supported power interfaces and values provided in the `enabled_power_interfaces` option has only one item: `ipmitool`. It is used.

console No default is configured. The `ipmi` hardware type supports the following console interfaces: `ipmitool-socat`, `ipmitool-shellinabox` and `no-console` (in this order). Of these three, only two are enabled: `no-console` and `ipmitool-shellinabox` (order does not matter). The intersection contains `ipmitool-shellinabox` and `no-console`. The first item is used, and it is `ipmitool-shellinabox`.

management Following the same calculation as *power*, the `ipmitool` management interface is used.

Hardware Inspection

The Bare Metal service supports hardware inspection that simplifies enrolling nodes - please see *Hardware Inspection* for details.

Tenant Networks and Port Groups

See *Multi-tenancy in the Bare Metal service* and *Port groups support*.

2.1.8 Using Bare Metal service as a standalone service

This guide explains how to configure and use the Bare Metal service standalone, i.e. without other OpenStack services. In this mode users are interacting with the bare metal API directly, not though OpenStack Compute.

Configuration

This guide covers manual configuration of the Bare Metal service in the standalone mode. Alternatively, Bifrost can be used for automatic configuration.

Service settings

It is possible to use the Bare Metal service without other OpenStack services. You should make the following changes to `/etc/ironic/ironic.conf`:

1. Choose an authentication strategy which supports standalone, one option is `noauth`:

   ```
   [DEFAULT]
   auth_strategy=noauth
   ```

   Another option is `http_basic` where the credentials are stored in an Apache htpasswd format file:

   ```
   [DEFAULT]
   auth_strategy=http_basic
   http_basic_auth_user_file=/etc/ironic/htpasswd
   ```

   Only the `bcrypt` format is supported, and the Apache *htpasswd* utility can be used to populate the file with entries, for example:

   ```
   htpasswd -nbB myName myPassword >> /etc/ironic/htpasswd
   ```

2. If you want to disable the Networking service, you should have your network pre-configured to serve DHCP and TFTP for machines that youre deploying. To disable it, change the following lines:

   ```
   [dhcp]
   dhcp_provider=none
   ```

   Note: If you disabled the Networking service and the driver that you use is supported by at most one conductor, PXE boot will still work for your nodes without any manual config editing. This is because you know all the DHCP options that will be used for deployment and can set up your DHCP server appropriately.

   If you have multiple conductors per driver, it would be better to use Networking since it will do all the dynamically changing configurations for you.

3. If you want to disable using a messaging broker between conductor and API processes, switch to JSON RPC instead:

   ```
   [DEFAULT]
   rpc_transport = json-rpc
   ```

   JSON RPC also has its own authentication strategy. If it is not specified then the stategy defaults to `[DEFAULT] auth_strategy`. The following will set JSON RPC to `noauth`:

```
[json_rpc]
auth_strategy = noauth
```

For `http_basic` the conductor server needs a credentials file to validate requests:

```
[json_rpc]
auth_strategy = http_basic
http_basic_auth_user_file = /etc/ironic/htpasswd-json-rpc
```

The API server also needs client-side credentials to be specified:

```
[json_rpc]
auth_type = http_basic
username = myName
password = myPassword
```

4. Starting with the Yoga release series, you can use a combined API+conductor service and completely disable the RPC. Set

```
[DEFAULT]
rpc_transport = none
```

and use the `ironic` executable to start the combined service.

Note: The combined service also works with RPC enabled, which can be useful for some deployments, but may not be advisable for all security models.

Using CLI

To use the baremetal CLI, set up these environment variables. If the `noauth` authentication strategy is being used, the value `none` must be set for OS_AUTH_TYPE. OS_ENDPOINT is the URL of the ironic-api process. For example:

```
export OS_AUTH_TYPE=none
export OS_ENDPOINT=http://localhost:6385/
```

If the `http_basic` authentication strategy is being used, the value `http_basic` must be set for OS_AUTH_TYPE. For example:

```
export OS_AUTH_TYPE=http_basic
export OS_ENDPOINT=http://localhost:6385/
export OS_USERNAME=myUser
export OS_PASSWORD=myPassword
```

Enrollment

Preparing images

If you dont use Image service, its possible to provide images to Bare Metal service via a URL.

At the moment, only two types of URLs are acceptable instead of Image service UUIDs: HTTP(S) URLs (for example, http://my.server.net/images/img) and file URLs (file:///images/img).

There are however some limitations for different hardware interfaces:

- If youre using *Direct deploy* with HTTP(s) URLs, you have to provide the Bare Metal service with the a checksum of your instance image.

  MD5 is used by default for backward compatibility reasons. To compute an MD5 checksum, you can use the following command:

  ```
  $ md5sum image.qcow2
  ed82def8730f394fb85aef8a208635f6  image.qcow2
  ```

  Alternatively, use a SHA256 checksum or any other algorithm supported by the Pythons hashlib, e.g.:

  ```
  $ sha256sum image.qcow2
  9f6c942ad81690a9926ff530629fb69a82db8b8ab267e2cbd59df417c1a28060  image.
  ↪qcow2
  ```

- *Direct deploy* started supporting `file://` images in the Victoria release cycle, before that only HTTP(s) had been supported.

  Warning: File images must be accessible to every conductor! Use a shared file system if you have more than one conductor. The ironic CLI tool will not transfer the file from a local machine to the conductor(s).

Note: The Bare Metal service tracks content changes for non-Glance images by checking their modification date and time. For example, for HTTP image, if Last-Modified header value from response to a HEAD request to http://my.server.net/images/deploy.ramdisk is greater than cached image modification time, Ironic will re-download the content. For file:// images, the file system modification time is used.

If the HTTP server does not provide the last modification date and time, the image will be redownloaded every time it is used.

Enrolling nodes

1. Create a node in Bare Metal service. At minimum, you must specify the driver name (for example, `ipmi`). You can also specify all the required driver parameters in one command. This will return the node UUID:

```
$ baremetal node create --driver ipmi \
    --driver-info ipmi_address=ipmi.server.net \
    --driver-info ipmi_username=user \
    --driver-info ipmi_password=pass \
    --driver-info deploy_kernel=file:///images/deploy.vmlinuz \
    --driver-info deploy_ramdisk=http://my.server.net/images/deploy.
↪ramdisk
+--------------+--------------------------------------------------------
↪----------------+
| Property     | Value                                                  ␣
↪                |
+--------------+--------------------------------------------------------
↪----------------+
| uuid         | be94df40-b80a-4f63-b92b-e9368ee8d14c                   ␣
↪                |
| driver_info  | {u'deploy_ramdisk': u'http://my.server.net/images/deploy.
↪ramdisk',       |
|              | u'deploy_kernel': u'file:///images/deploy.vmlinuz', u
↪'ipmi_address':     |
|              | u'ipmi.server.net', u'ipmi_username': u'user', u'ipmi_
↪password':          |
|              | u'******'}                                             ␣
↪                |
| extra        | {}                                                     ␣
↪                |
| driver       | ipmi                                                   ␣
↪                |
| chassis_uuid |                                                        ␣
↪                |
| properties   | {}                                                     ␣
↪                |
+--------------+--------------------------------------------------------
↪----------------+
```

   Note that here deploy_kernel and deploy_ramdisk contain links to images instead of Image service UUIDs.

2. As in case of Compute service, you can also provide `capabilities` to node properties, but they will be used only by Bare Metal service (for example, boot mode). Although you dont need to add properties like `memory_mb`, `cpus` etc. as Bare Metal service will require UUID of a node youre going to deploy.

3. Then create a port to inform Bare Metal service of the network interface cards which are part of the node by creating a port with each NICs MAC address. In this case, theyre used for naming of PXE configs for a node:

```
baremetal port create $MAC_ADDRESS --node $NODE_UUID
```

Once the installation is done, please see *Deploying with Bare Metal service* for information on how to deploy bare metal machines.

Deploying

The content has been migrated, please see *Deploying with Bare Metal service*.

2.1.9 Enabling the configuration drive (configdrive)

The Bare Metal service supports exposing a configuration drive image to the instances.

The configuration drive is used to store instance-specific metadata and is present to the instance as a disk partition labeled `config-2`. The configuration drive has a maximum size of 64MB. One use case for using the configuration drive is to expose a networking configuration when you do not use DHCP to assign IP addresses to instances.

The configuration drive is usually used in conjunction with the Compute service, but the Bare Metal service also offers a standalone way of using it. The following sections will describe both methods.

When used with Compute service

To enable the configuration drive for a specific request, pass `--config-drive true` parameter to the **`nova boot`** command, for example:

```
nova boot --config-drive true --flavor baremetal --image test-image instance-1
```

Its also possible to enable the configuration drive automatically on all instances by configuring the `OpenStack Compute service` to always create a configuration drive by setting the following option in the `/etc/nova/nova.conf` file, for example:

```
[DEFAULT]
...

force_config_drive=True
```

In some cases, you may wish to pass a user customized script when deploying an instance. To do this, pass `--user-data /path/to/file` to the **`nova boot`** command.

When used standalone

When used without the Compute service, the operator needs to create a configuration drive and provide the file or HTTP URL to the Bare Metal service. See *Deploying with a config drive* for details.

Configuration drive storage in an object store

Under normal circumstances, the configuration drive can be stored in the Bare Metal service when the size is less than 64KB. Optionally, if the size is larger than 64KB there is support to store it in a swift endpoint. Both swift and radosgw use swift-style APIs.

The following option in `/etc/ironic/ironic.conf` enables swift as an object store backend to store config drive. This uses the Identity service to establish a session between the Bare Metal service and the Object Storage service.

```
[deploy]
...

configdrive_use_object_store = True
```

Use the following options in `/etc/ironic/ironic.conf` to enable radosgw. Credentials in the swift section are needed because radosgw will not use the Identity service and relies on radosgws username and password authentication instead.

```
[deploy]
...

configdrive_use_object_store = True

[swift]
...

username = USERNAME
password = PASSWORD
auth_url = http://RADOSGW_IP:8000/auth/v1
```

If the *Direct deploy* is being used, edit `/etc/glance/glance-api.conf` to store the instance images in respective object store (radosgw or swift) as well:

```
[glance_store]
...

swift_store_user = USERNAME
swift_store_key = PASSWORD
swift_store_auth_address = http://RADOSGW_OR_SWIFT_IP:PORT/auth/v1
```

Accessing the configuration drive data

When the configuration drive is enabled, the Bare Metal service will create a partition on the instance disk and write the configuration drive image onto it. The configuration drive must be mounted before use. This is performed automatically by many tools, such as cloud-init and cloudbase-init. To mount it manually on a Linux distribution that supports accessing devices by labels, simply run the following:

```
mkdir -p /mnt/config
mount /dev/disk/by-label/config-2 /mnt/config
```

If the guest OS doesnt support accessing devices by labels, you can use other tools such as `blkid` to identify which device corresponds to the configuration drive and mount it, for example:

```
CONFIG_DEV=$(blkid -t LABEL="config-2" -odevice)
mkdir -p /mnt/config
mount $CONFIG_DEV /mnt/config
```

Cloud-init integration

The configuration drive can be especially useful when used with cloud-init, but in order to use it we should follow some rules:

- `Cloud-init` data should be organized in the expected format.
- Since the Bare Metal service uses a disk partition as the configuration drive, it will only work with cloud-init version >= 0.7.5.
- `Cloud-init` has a collection of data source modules, so when building the image with disk-image-builder we have to define `DIB_CLOUD_INIT_DATASOURCES` environment variable and set the appropriate sources to enable the configuration drive, for example:

  ```
  DIB_CLOUD_INIT_DATASOURCES="ConfigDrive, OpenStack" disk-image-create -o␣
  ↪fedora-cloud-image fedora baremetal
  ```

  For more information see how to configure cloud-init data sources.

2.1.10 Advanced features

Local boot with partition images

The Bare Metal service supports local boot with partition images, meaning that after the deployment the nodes subsequent reboots wont happen via PXE or Virtual Media. Instead, it will boot from a local boot loader installed on the disk.

Note: Whole disk images, on the contrary, support only local boot, and use it by default.

Its important to note that in order for this to work the image being deployed with Bare Metal service **must** contain `grub2` installed within it.

Enabling the local boot is different when Bare Metal service is used with Compute service and without it. The following sections will describe both methods.

Enabling local boot with Compute service

To enable local boot we need to set a capability on the bare metal node, for example:

```
baremetal node set <node-uuid> --property capabilities="boot_option:local"
```

Nodes having `boot_option` set to `local` may be requested by adding an `extra_spec` to the Compute service flavor, for example:

```
nova flavor-key baremetal set capabilities:boot_option="local"
```

Note: If the node is configured to use `UEFI`, Bare Metal service will create an `EFI partition` on the disk and switch the partition table format to `gpt`. The `EFI partition` will be used later by the boot loader (which is installed from the deploy ramdisk).

Enabling local boot without Compute

Since adding `capabilities` to the nodes properties is only used by the nova scheduler to perform more advanced scheduling of instances, we need a way to enable local boot when Compute is not present. To do that we can simply specify the capability via the `instance_info` attribute of the node, for example:

```
baremetal node set <node-uuid> --instance-info capabilities='{"boot_option":
↪"local"}'
```

Specifying the disk for deployment (root device hints)

The Bare Metal service supports passing hints to the deploy ramdisk about which disk it should pick for the deployment. The list of supported hints is:

- model (STRING): device identifier
- vendor (STRING): device vendor
- serial (STRING): disk serial number
- size (INT): size of the device in GiB

  Note: A nodes local_gb property is often set to a value 1 GiB less than the actual disk size to account for partitioning (this is how DevStack, TripleO and Ironic Inspector work, to name a few). However, in this case `size` should be the actual size. For example, for a 128 GiB disk `local_gb` will be 127, but size hint will be 128.

- wwn (STRING): unique storage identifier
- wwn_with_extension (STRING): unique storage identifier with the vendor extension appended
- wwn_vendor_extension (STRING): unique vendor storage identifier

- rotational (BOOLEAN): whether its a rotational device or not. This hint makes it easier to distinguish HDDs (rotational) and SSDs (not rotational) when choosing which disk Ironic should deploy the image onto.
- hctl (STRING): the SCSI address (Host, Channel, Target and Lun), e.g 1:0:0:0
- by_path (STRING): the alternate device name corresponding to a particular PCI or iSCSI path, e.g /dev/disk/by-path/pci-0000:00
- name (STRING): the device name, e.g /dev/md0

> **Warning:** The root device hint name should only be used for devices with constant names (e.g RAID volumes). For SATA, SCSI and IDE disk controllers this hint is not recommended because the order in which the device nodes are added in Linux is arbitrary, resulting in devices like /dev/sda and /dev/sdb switching around at boot time.

To associate one or more hints with a node, update the nodes properties with a `root_device` key, for example:

```
baremetal node set <node-uuid> --property root_device='{"wwn":
↪"0x4000cca77fc4dba1"}'
```

That will guarantee that Bare Metal service will pick the disk device that has the `wwn` equal to the specified wwn value, or fail the deployment if it can not be found.

Note: Starting with the Ussuri release, root device hints can be specified per-instance, see *Using Bare Metal service as a standalone service*.

The hints can have an operator at the beginning of the value string. If no operator is specified the default is == (for numerical values) and `s==` (for string values). The supported operators are:

- For numerical values:
  - = equal to or greater than. This is equivalent to >= and is supported for legacy reasons
  - == equal to
  - != not equal to
  - >= greater than or equal to
  - > greater than
  - <= less than or equal to
  - < less than
- For strings (as python comparisons):
  - `s==` equal to
  - `s!=` not equal to
  - `s>=` greater than or equal to
  - `s>` greater than
  - `s<=` less than or equal to

  - s< less than
  - <in> substring
- For collections:
  - <all-in> all elements contained in collection
  - <or> find one of these

Examples are:

- Finding a disk larger or equal to 60 GiB and non-rotational (SSD):

```
baremetal node set <node-uuid> --property root_device='{"size": ">= 60",
↪"rotational": false}'
```

- Finding a disk whose vendor is `samsung` or `winsys`:

```
baremetal node set <node-uuid> --property root_device='{"vendor": "<or>␣
↪samsung <or> winsys"}'
```

Note: If multiple hints are specified, a device must satisfy all the hints.

Appending kernel parameters to boot instances

The Bare Metal service supports passing custom kernel parameters to boot instances to fit users requirements. The way to append the kernel parameters is depending on how to boot instances.

Network boot

Currently, the Bare Metal service supports assigning unified kernel parameters to PXE booted instances by:

- Modifying the `[pxe]/kernel_append_params` configuration option, for example:

```
[pxe]
kernel_append_params = quiet splash
```

Note: The option was called `pxe_append_params` before the Xena cycle.

- Copying a template from shipped templates to another place, for example:

```
https://opendev.org/openstack/ironic/src/branch/master/ironic/drivers/
↪modules/pxe_config.template
```

Making the modifications and pointing to the custom template via the configuration options: `[pxe]/pxe_config_template` and `[pxe]/uefi_pxe_config_template`.

Local boot

For local boot instances, users can make use of configuration drive (see *Enabling the configuration drive (configdrive)*) to pass a custom script to append kernel parameters when creating an instance. This is more flexible and can vary per instance. Here is an example for grub2 with ubuntu, users can customize it to fit their use case:

```
#!/usr/bin/env python
import os

# Default grub2 config file in Ubuntu
grub_file = '/etc/default/grub'
# Add parameters here to pass to instance.
kernel_parameters = ['quiet', 'splash']
grub_cmd = 'GRUB_CMDLINE_LINUX'
old_grub_file = grub_file+'~'
os.rename(grub_file, old_grub_file)
cmdline_existed = False
with open(grub_file, 'w') as writer, \
       open(old_grub_file, 'r') as reader:
       for line in reader:
           key = line.split('=')[0]
           if key == grub_cmd:
               #If there is already some value:
               if line.strip()[-1] == '"':
                   line = line.strip()[:-1] + ' ' + ' '.join(kernel_
↪parameters) + '"'
               cmdline_existed = True
           writer.write(line)
       if not cmdline_existed:
           line = grub_cmd + '=' + '"' + ' '.join(kernel_parameters) + '"'
           writer.write(line)

os.remove(old_grub_file)
os.system('update-grub')
os.system('reboot')
```

Console

In order to change default console configuration in the Bare Metal service configuration file (`[pxe]` section in `/etc/ironic/ironic.conf`), include the serial port terminal and serial speed. Serial speed must be the same as the serial configuration in the BIOS settings, so that the operating system boot process can be seen in the serial console or web console. Following examples represent possible parameters for serial and web console respectively.

- Node serial console. The console parameter `console=ttyS0,115200n8` uses `ttyS0` for console output at `115200bps, 8bit, non-parity`, e.g.:

```
[pxe]
```

(continues on next page)

(continued from previous page)

```
# Additional append parameters for baremetal PXE boot.
kernel_append_params = nofb nomodeset vga=normal console=ttyS0,115200n8
```

- For node web console configuration is similar with the addition of `ttyX` parameter, see example:

```
[pxe]

# Additional append parameters for baremetal PXE boot.
kernel_append_params = nofb nomodeset vga=normal console=tty0␣
↪console=ttyS0,115200n8
```

For detailed information on how to add consoles see the reference documents kernel params and serial console. In case of local boot the Bare Metal service is not able to control kernel boot parameters. To configure console locally, follow Local boot section above.

Boot mode support

Some of the bare metal hardware types (namely, `redfish`, `ilo` and generic `ipmi`) support setting boot mode (Legacy BIOS or UEFI).

Note: Setting boot mode support in generic `ipmi` driver is coupled with setting boot device. That makes boot mode support in the `ipmi` driver incomplete.

Note: In this chapter we will distinguish *ironic node* from *bare metal node*. The difference is that *ironic node* refers to a logical node, as it is configured in ironic, while *bare metal node* indicates the hardware machine that ironic is managing.

The following rules apply in order when ironic manages node boot mode:

- If the hardware type (or bare metal node) does not implement reading current boot mode of the bare metal node, then ironic assumes that boot mode is not set on the bare metal node
- If boot mode is not set on ironic node and bare metal node boot mode is unknown (not set, cant be read etc.), ironic node boot mode is set to the value of the *[deploy]/default_boot_mode* option
- If boot mode is set on a bare metal node, but is not set on ironic node, bare metal node boot mode is set on ironic node
- If boot mode is set on ironic node, but is not set on the bare metal node, ironic node boot mode is attempted to be set on the bare metal node (failure to set boot mode on the bare metal node will not fail ironic node deployment)
- If different boot modes appear on to be set ironic node and on the bare metal node, ironic node boot mode is attempted to be set on the bare metal node (failure to set boot mode on the bare metal node will fail ironic node deployment)

Warning: If a bare metal node does not support setting boot mode, then the operator needs to make sure that boot mode configuration is consistent between ironic node and the bare metal node.

The boot modes can be configured in the Bare Metal service in the following way:

- Only one boot mode (either `uefi` or `bios`) can be configured for the node.
- If the operator wants a node to boot always in `uefi` mode or `bios` mode, then they may use `capabilities` parameter within `properties` field of an bare metal node. The operator must manually set the appropriate boot mode on the bare metal node.

  To configure a node in `uefi` mode, then set `capabilities` as below:

  ```
  openstack baremetal node set <node-uuid> --property capabilities='boot_
  ↪mode:uefi'
  ```

  Conversely, to configure a node in `bios` mode, then set the `capabilities` as below:

  ```
  openstack baremetal node set <node-uuid> --property capabilities='boot_
  ↪mode:bios'
  ```

Note:

The Ironic project changed the default boot mode setting for nodes from `bios` to `uefi` during the Yoga development cycle.

Nodes having `boot_mode` set to `uefi` may be requested by adding an `extra_spec` to the Compute service flavor:

```
nova flavor-key ironic-test-3 set capabilities:boot_mode="uefi"
nova boot --flavor ironic-test-3 --image test-image instance-1
```

If `capabilities` is used in `extra_spec` as above, nova scheduler (`ComputeCapabilitiesFilter`) will match only bare metal nodes which have the `boot_mode` set appropriately in `properties/capabilities`. It will filter out rest of the nodes.

The above facility for matching in the Compute service can be used in heterogeneous environments where there is a mix of `uefi` and `bios` machines, and operator wants to provide a choice to the user regarding boot modes. If the flavor doesnt contain `boot_mode` and `boot_mode` is configured for bare metal nodes, then nova scheduler will consider all nodes and user may get either `bios` or `uefi` machine.

Some hardware support setting secure boot mode, see *UEFI secure boot mode* for details.

Choosing the disk label

Note: The term `disk label` is historically used in Ironic and was taken from parted. Apparently everyone seems to have a different word for `disk label` - these are all the same thing: disk type, partition table, partition map and so on

---

Ironic allows operators to choose which disk label they want their bare metal node to be deployed with when Ironic is responsible for partitioning the disk; therefore choosing the disk label does not apply when the image being deployed is a `whole disk image`.

There are some edge cases where someone may want to choose a specific disk label for the images being deployed, including but not limited to:

- For machines in `bios` boot mode with disks larger than 2 terabytes its recommended to use a `gpt` disk label. Thats because a capacity beyond 2 terabytes is not addressable by using the MBR partitioning type. But, although GPT claims to be backward compatible with legacy BIOS systems thats not always the case.
- Operators may want to force the partitioning to be always MBR (even if the machine is deployed with boot mode `uefi`) to avoid breakage of applications and tools running on those instances.

The disk label can be configured in two ways; when Ironic is used with the Compute service or in standalone mode. The following bullet points and sections will describe both methods:

- When no disk label is provided Ironic will configure it according to the boot mode (see *Boot mode support*); `bios` boot mode will use `msdos` and `uefi` boot mode will use `gpt`.
- Only one disk label - either `msdos` or `gpt` - can be configured for the node.

When used with Compute service

When Ironic is used with the Compute service the disk label should be set to nodes `properties/capabilities` field and also to the flavor which will request such capability, for example:

```
baremetal node set <node-uuid> --property capabilities='disk_label:gpt'
```

As for the flavor:

```
nova flavor-key baremetal set capabilities:disk_label="gpt"
```

When used in standalone mode

When used without the Compute service, the disk label should be set directly to the nodes `instance_info` field, as below:

```
baremetal node set <node-uuid> --instance-info capabilities='{"disk_label":
↪"gpt"}'
```

Notifications

The Bare Metal service supports the emission of notifications, which are messages sent on a message broker (like RabbitMQ or anything else supported by the oslo messaging library) that indicate various events which occur, such as when a node changes power states. These can be consumed by an external service reading from the message bus. For example, Searchlight is an OpenStack service that uses notifications to index (and make searchable) resources from the Bare Metal service.

Notifications are disabled by default. For a complete list of available notifications and instructions for how to enable them, see the *Notifications*.

Configuring node web console

See *Configuring Web or Serial Console*.

2.1.11 Troubleshooting

Once all the services are running and configured properly, and a node has been enrolled with the Bare Metal service and is in the `available` provision state, the Compute service should detect the node as an available resource and expose it to the scheduler.

Note: There is a delay, and it may take up to a minute (one periodic task cycle) for the Compute service to recognize any changes in the Bare Metal services resources (both additions and deletions).

In addition to watching `nova-compute` log files, you can see the available resources by looking at the list of Compute hypervisors. The resources reported therein should match the bare metal node properties, and the Compute service flavor.

Here is an example set of commands to compare the resources in Compute service and Bare Metal service:

```
$ baremetal node list
+--------------------------------------+---------------+-------------+--------
↪------------+-------------+
| UUID                                 | Instance UUID | Power State |␣
↪Provisioning State | Maintenance |
+--------------------------------------+---------------+-------------+--------
↪------------+-------------+
| 86a2b1bb-8b29-4964-a817-f90031debddb | None          | power off   |␣
↪available          | False       |
+--------------------------------------+---------------+-------------+--------
↪------------+-------------+

$ baremetal node show 86a2b1bb-8b29-4964-a817-f90031debddb
+------------------------+------------------------------------------------------
↪------------------+
| Property               | Value                                                ␣
↪                  |
+------------------------+------------------------------------------------------
↪------------------+
| instance_uuid          | None                                                 ␣
↪                  |
| properties             | {u'memory_mb': u'1024', u'cpu_arch': u'x86_64', u
↪'local_gb': u'10',  |
|                        | u'cpus': u'1'}                                       ␣
↪                  |
| maintenance            | False                                                ␣
↪                  |
| driver_info            | { [SNIP] }                                           ␣
↪                  |
| extra                  | {}                                                   ␣
↪                  |
```

(continues on next page)

(continued from previous page)

```
| last_error             | None                                                 ␣
↪                    |
| created_at             | 2014-11-20T23:57:03+00:00                            ␣
↪                    |
| target_provision_state | None                                                 ␣
↪                    |
| driver                 | ipmi                                                 ␣
↪                    |
| updated_at             | 2014-11-21T00:47:34+00:00                            ␣
↪                    |
| instance_info          | {}                                                   ␣
↪                    |
| chassis_uuid           | 7b49bbc5-2eb7-4269-b6ea-3f1a51448a59                 ␣
↪                    |
| provision_state        | available                                            ␣
↪                    |
| reservation            | None                                                 ␣
↪                    |
| power_state            | power off                                            ␣
↪                    |
| console_enabled        | False                                                ␣
↪                    |
| uuid                   | 86a2b1bb-8b29-4964-a817-f90031debddb                 ␣
↪                    |
+------------------------+------------------------------------------------------
↪------------------+

$ nova hypervisor-list
+--------------------------------------+---------------------------------------
↪+-------+---------+
| ID                                   | Hypervisor hostname                  ␣
↪| State | Status  |
+--------------------------------------+---------------------------------------
↪+-------+---------+
| 584cfdc8-9afd-4fbb-82ef-9ff25e1ad3f3 | 86a2b1bb-8b29-4964-a817-f90031debddb␣
↪| up    | enabled |
+--------------------------------------+---------------------------------------
↪+-------+---------+

$ nova hypervisor-show 584cfdc8-9afd-4fbb-82ef-9ff25e1ad3f3
+-------------------------+--------------------------------------+
| Property                | Value                                |
+-------------------------+--------------------------------------+
| cpu_info                | baremetal cpu                        |
| current_workload        | 0                                    |
| disk_available_least    | -                                    |
| free_disk_gb            | 10                                   |
| free_ram_mb             | 1024                                 |
| host_ip                 | [ SNIP ]                             |
```

(continues on next page)

(continued from previous page)

```
| hypervisor_hostname     | 86a2b1bb-8b29-4964-a817-f90031debddb |
| hypervisor_type         | ironic                               |
| hypervisor_version      | 1                                    |
| id                      | 1                                    |
| local_gb                | 10                                   |
| local_gb_used           | 0                                    |
| memory_mb               | 1024                                 |
| memory_mb_used          | 0                                    |
| running_vms             | 0                                    |
| service_disabled_reason | -                                    |
| service_host            | my-test-host                         |
| service_id              | 6                                    |
| state                   | up                                   |
| status                  | enabled                              |
| vcpus                   | 1                                    |
| vcpus_used              | 0                                    |
+-------------------------+--------------------------------------+
```

Maintenance mode

Maintenance mode may be used if you need to take a node out of the resource pool. Putting a node in maintenance mode will prevent Bare Metal service from executing periodic tasks associated with the node. This will also prevent Compute service from placing a tenant instance on the node by not exposing the node to the nova scheduler. Nodes can be placed into maintenance mode with the following command.

```
$ baremetal node maintenance set $NODE_UUID
```

A maintenance reason may be included with the optional `--reason` command line option. This is a free form text field that will be displayed in the `maintenance_reason` section of the `node show` command.

```
$ baremetal node maintenance set $UUID --reason "Need to add ram."

$ baremetal node show $UUID

+------------------------+--------------------------------------+
| Property               | Value                                |
+------------------------+--------------------------------------+
| target_power_state     | None                                 |
| extra                  | {}                                   |
| last_error             | None                                 |
| updated_at             | 2015-04-27T15:43:58+00:00            |
| maintenance_reason     | Need to add ram.                     |
| ...                    | ...                                  |
| maintenance            | True                                 |
| ...                    | ...                                  |
+------------------------+--------------------------------------+
```

To remove maintenance mode and clear any `maintenance_reason` use the following command.

```
$ baremetal node maintenance unset $NODE_UUID
```

2.1.12 Next steps

Your OpenStack environment now includes the Bare Metal service.

2.1.13 Create user images for the Bare Metal service

The content has been migrated, please see *Creating instance images*.

2.2 Bare Metal Service Upgrade Guide

This document outlines various steps and notes for operators to consider when upgrading their ironic-driven clouds from previous versions of OpenStack.

The Bare Metal (ironic) service is tightly coupled with the ironic driver that is shipped with the Compute (nova) service. Some special considerations must be taken into account when upgrading your cloud.

Both offline and rolling upgrades are supported.

2.2.1 Plan your upgrade

- Rolling upgrades are available starting with the Pike release; that is, when upgrading from Ocata. This means that it is possible to do an upgrade with minimal to no downtime of the Bare Metal API.
- Upgrades are only supported between two consecutive named releases. This means that you cannot upgrade Ocata directly into Queens; you need to upgrade into Pike first.
- The release notes should always be read carefully when upgrading the Bare Metal service. Specific upgrade steps and considerations are documented there.
- The Bare Metal service should always be upgraded before the Compute service.

  Note: The ironic virt driver in nova always uses a specific version of the ironic REST API. This API version may be one that was introduced in the same development cycle, so upgrading nova first may result in nova being unable to use the Bare Metal API.

- Make a backup of your database. Ironic does not support downgrading of the database. Hence, in case of upgrade failure, restoring the database from a backup is the only choice.
- Before starting your upgrade, it is best to ensure that all nodes have reached, or are in, a stable `provision_state`. Nodes in states with long running processes such as deploying or cleaning, may fail, and may require manual intervention to return them to the available hardware pool. This is most likely in cases where a timeout has occurred or a service was terminated abruptly. For a visual diagram detailing states and possible state transitions, please see *Bare Metal State Machine*.

2.2.2 Offline upgrades

In an offline (or cold) upgrade, the Bare Metal service is not available during the upgrade, because all the services have to be taken down.

When upgrading the Bare Metal service, the following steps should always be taken in this order:

1. upgrade the ironic-python-agent image
2. update ironic code, without restarting services
3. run database schema migrations via `ironic-dbsync upgrade`
4. restart ironic-conductor and ironic-api services

Once the above is done, do the following:

- update any applicable configuration options to stop using any deprecated features or options, and perform any required work to transition to alternatives. All the deprecated features and options will be supported for one release cycle, so should be removed before your next upgrade is performed.
- upgrade python-ironicclient along with any other services connecting to the Bare Metal service as a client, such as nova-compute
- run the `ironic-dbsync online_data_migrations` command to make sure that data migrations are applied. The command lets you limit the impact of the data migrations with the `--max-count` option, which limits the number of migrations executed in one run. You should complete all of the migrations as soon as possible after the upgrade.

> **Warning:** You will not be able to start an upgrade to the release after this one, until this has been completed for the current release. For example, as part of upgrading from Ocata to Pike, you need to complete Pikes data migrations. If this not done, you will not be able to upgrade to Queens it will not be possible to execute Queens database schema updates.

2.2.3 Rolling upgrades

To Reduce downtime, the services can be upgraded in a rolling fashion, meaning to upgrade one or a few services at a time to minimize impact.

Rolling upgrades are available starting with the Pike release. This feature makes it possible to upgrade between releases, such as Ocata to Pike, with minimal to no downtime of the Bare Metal API.

Requirements

To facilitate an upgrade in a rolling fashion, you need to have a highly-available deployment consisting of at least two ironic-api and two ironic-conductor services. Use of a load balancer to balance requests across the ironic-api services is recommended, as it allows for a minimal impact to end users.

Concepts

There are four aspects of the rolling upgrade process to keep in mind:

- API and RPC version pinning, and versioned object backports
- online data migrations
- graceful service shutdown
- API load balancer draining

API & RPC version pinning and versioned object backports

Through careful RPC versioning, newer services are able to talk to older services (and vice-versa). The `[DEFAULT]/pin_release_version` configuration option is used for this. It should be set (pinned) to the release version that the older services are using. The newer services will backport RPC calls and objects to their appropriate versions from the pinned release. If the `IncompatibleObjectVersion` exception occurs, it is most likely due to an incorrect or unspecified `[DEFAULT]/pin_release_version` configuration value. For example, when `[DEFAULT]/pin_release_version` is not set to the older release version, no conversion will happen during the upgrade.

For the ironic-api service, the API version is pinned via the same `[DEFAULT]/pin_release_version` configuration option as above. When pinned, the new ironic-api services will not service any API requests with Bare Metal API versions that are higher than what the old ironic-api services support. HTTP status code 406 is returned for such requests. This prevents new features (available in new API versions) from being used until after the upgrade has been completed.

Online data migrations

To make database schema migrations less painful to execute, we have implemented process changes to facilitate upgrades.

- All data migrations are banned from schema migration scripts.
- Schema migration scripts only update the database schema.
- Data migrations must be done at the end of the rolling upgrade process, after the schema migration and after the services have been upgraded to the latest release.

All data migrations are performed using the `ironic-dbsync online_data_migrations` command. It can be run as a background process so that it does not interrupt running services; however it must be run to completion for a cold upgrade if the intent is to make use of new features immediately.

(You would also execute the same command with services turned off if you are doing a cold upgrade).

This data migration must be completed. If not, you will not be able to upgrade to future releases. For example, if you had upgraded from Ocata to Pike but did not do the data migrations, you will not be able to upgrade from Pike to Queens. (More precisely, you will not be able to apply Queens schema migrations.)

Graceful conductor service shutdown

The ironic-conductor service is a Python process listening for messages on a message queue. When the operator sends the SIGTERM signal to the process, the service stops consuming messages from the queue, so that no additional work is picked up. It completes any outstanding work and then terminates. During this process, messages can be left on the queue and will be processed after the Python process starts back up. This gives us a way to shutdown a service using older code, and start up a service using newer code with minimal impact.

Note: This was tested with RabbitMQ messaging backend and may vary with other backends.

Nodes that are being acted upon by an ironic-conductor process, which are not in a stable state, may encounter failures. Node failures that occur during an upgrade are likely due to timeouts, resulting from delays involving messages being processed and acted upon by a conductor during long running, multi-step processes such as deployment or cleaning.

API load balancer draining

If you are using a load balancer for the ironic-api services, we recommend that you redirect requests to the new API services and drain off of the ironic-api services that have not yet been upgraded.

Rolling upgrade process

Before maintenance window

- Upgrade the ironic-python-agent image
- Using the new release (ironic code), execute the required database schema updates by running the database upgrade command: `ironic-dbsync upgrade`. These schema change operations should have minimal or no effect on performance, and should not cause any operations to fail (but please check the release notes). You can:
  - install the new release on an existing system
  - install the new release in a new virtualenv or a container

  At this point, new columns and tables may exist in the database. These database schema changes are done in a way that both the old and new (N and N+1) releases can perform operations against the same schema.

Note: Ironic bases its API, RPC and object storage format versions on the `[DEFAULT]/pin_release_version` configuration option. It is advisable to automate the deployment of changes in configuration files to make the process less error prone and repeatable.

During maintenance window

1. All ironic-conductor services should be upgraded first. Ensure that at least one ironic-conductor service is running at all times. For every ironic-conductor, either one by one or a few at a time:
   - shut down the service. Messages from the ironic-api services to the conductors are load-balanced by the message queue and a hash-ring, so the only thing you need to worry about is to shut the service down gracefully (using `SIGTERM` signal) to make sure it will finish all the requests being processed before shutting down.
   - upgrade the installed version of ironic and dependencies
   - set the `[DEFAULT]/pin_release_version` configuration option value to the version you are upgrading from (that is, the old version). Based on this setting, the new ironic-conductor services will downgrade any RPC communication and data objects to conform to the old service. For example, if you are upgrading from Ocata to Pike, set this value to `ocata`.
   - start the service
2. The next service to upgrade is ironic-api. Ensure that at least one ironic-api service is running at all times. You may want to start another temporary instance of the older ironic-api to handle the load while you are upgrading the original ironic-api services. For every ironic-api service, either one by one or a few at a time:
   - in HA deployment you are typically running them behind a load balancer (for example HAProxy), so you need to take the service instance out of the balancer
   - shut it down
   - upgrade the installed version of ironic and dependencies
   - set the `[DEFAULT]/pin_release_version` configuration option value to the version you are upgrading from (that is, the old version). Based on this setting, the new ironic-api services will downgrade any RPC communication and data objects to conform to the old service. In addition, the new services will return HTTP status code 406 for any requests with newer API versions that the old services did not support. This prevents new features (available in new API versions) from being used until after the upgrade has been completed. For example, if you are upgrading from Ocata to Pike, set this value to `ocata`.
   - restart the service
   - add it back into the load balancer

   After upgrading all the ironic-api services, the Bare Metal service is running in the new version but with downgraded RPC communication and database object storage formats. New features (in new API versions) are not supported, because they could fail when objects are in the downgraded object formats and some internal RPC API functions may still not be available.
3. For all the ironic-conductor services, one at a time:
   - remove the `[DEFAULT]/pin_release_version` configuration option setting
   - restart the ironic-conductor service
4. For all the ironic-api services, one at a time:
   - remove the `[DEFAULT]/pin_release_version` configuration option setting
   - restart the ironic-api service

After maintenance window

Now that all the services are upgraded, the system is able to use the latest version of the RPC protocol and able to access all the features of the new release.

- Update any applicable configuration options to stop using any deprecated features or options, and perform any required work to transition to alternatives. All the deprecated features and options will be supported for one release cycle, so should be removed before your next upgrade is performed.
- Upgrade `python-ironicclient` along with other services connecting to the Bare Metal service as a client, such as `nova-compute`.

> **Warning:** A `nova-compute` instance tries to attach VIFs to all active instances on start up. Make sure that for all active nodes there is at least one running `ironic-conductor` process to manage them. Otherwise the instances will be moved to the `ERROR` state on the `nova-compute` start up.

- Run the `ironic-dbsync online_data_migrations` command to make sure that data migrations are applied. The command lets you limit the impact of the data migrations with the `--max-count` option, which limits the number of migrations executed in one run. You should complete all of the migrations as soon as possible after the upgrade.

> **Warning:** Note that you will not be able to start an upgrade to the next release after this one, until this has been completed for the current release. For example, as part of upgrading from Ocata to Pike, you need to complete Pikes data migrations. If this not done, you will not be able to upgrade to Queens it will not be possible to execute Queens database schema updates.

Upgrading to Hardware Types

Starting with the Rocky release, the Bare Metal service does not support *classic drivers* any more. If you still use *classic drivers*, please upgrade to *hardware types* immediately. Please see *Enabling drivers and hardware types* for details on *hardware types* and *hardware interfaces*.

Planning the upgrade

It is necessary to figure out which hardware types and hardware interfaces correspond to which classic drivers used in your deployment. The following table lists the classic drivers with their corresponding hardware types and the boot, deploy, inspect, management, and power hardware interfaces:

Classic Driver	Hardware Type	Boot	Deploy	Inspect	Management	Power
agent_ilo	ilo	ilo-virtual-media	direct	ilo	ilo	ilo
agent_ipmitool	ipmi	pxe	direct	inspector	ipmitool	ipmitool
agent_ipmitool_socat	ipmi	pxe	direct	inspector	ipmitool	ipmitool
agent_irmc	irmc	irmc-virtual-media	direct	irmc	irmc	irmc
iscsi_ilo	ilo	ilo-virtual-media	iscsi	ilo	ilo	ilo
iscsi_irmc	irmc	irmc-virtual-media	iscsi	irmc	irmc	irmc
pxe_drac	idrac	pxe	iscsi	idrac	idrac	idrac
pxe_drac_inspector	idrac	pxe	iscsi	inspector	idrac	idrac
pxe_ilo	ilo	ilo-pxe	iscsi	ilo	ilo	ilo
pxe_ipmitool	ipmi	pxe	iscsi	inspector	ipmitool	ipmitool
pxe_ipmitool_socat	ipmi	pxe	iscsi	inspector	ipmitool	ipmitool
pxe_irmc	irmc	irmc-pxe	iscsi	irmc	irmc	irmc
pxe_snmp	snmp	pxe	iscsi	no-inspect	fake	snmp

Note: The `inspector` *inspect* interface was only used if explicitly enabled in the configuration. Otherwise, `no-inspect` was used.

Note: `pxe_ipmitool_socat` and `agent_ipmitool_socat` use `ipmitool-socat` *console* interface (the default for the `ipmi` hardware type), while `pxe_ipmitool` and `agent_ipmitool` use `ipmitool-shellinabox`. See *Console* for details.

For out-of-tree drivers you may need to reach out to their maintainers or figure out the appropriate interfaces by researching the source code.

Configuration

You will need to enable hardware types and interfaces that correspond to your currently enabled classic drivers. For example, if you have the following configuration in your `ironic.conf`:

```
[DEFAULT]
enabled_drivers = pxe_ipmitool,agent_ipmitool
```

You will have to add this configuration as well:

```
[DEFAULT]
enabled_hardware_types = ipmi
enabled_boot_interfaces = pxe
enabled_deploy_interfaces = iscsi,direct
enabled_management_interfaces = ipmitool
enabled_power_interfaces = ipmitool
```

Note: For every interface type there is an option `default_<INTERFACE>_interface`, where `<INTERFACE>` is the interface type name. For example, one can make all nodes use the `direct` deploy method by default by setting:

```
[DEFAULT]
default_deploy_interface = direct
```

Migrating nodes

After the required items are enabled in the configuration, each nodes `driver` field has to be updated to a new value. You may need to also set new values for some or all interfaces:

```
export OS_BAREMETAL_API_VERSION=1.31

for uuid in $(baremetal node list --driver pxe_ipmitool -f value -c UUID); do
    baremetal node set <node> --driver ipmi --deploy-interface iscsi
done

for uuid in $(baremetal node list --driver agent_ipmitool -f value -c UUID);␣
↪do
    baremetal node set <node> --driver ipmi --deploy-interface direct
done
```

See *Enrollment* for more details on setting hardware types and interfaces.

Warning: It is not recommended to change the interfaces for `active` nodes. If absolutely needed, the nodes have to be put in the maintenance mode first:

```
baremetal node maintenance set $UUID \
    --reason "Changing driver and/or hardware interfaces"
```

```
# do the update, validate its correctness
baremetal node maintenance unset $UUID
```

Other interfaces

Care has to be taken to migrate from classic drivers using non-default interfaces. This chapter covers a few of the most commonly used.

Ironic Inspector

Some classic drivers, notably `pxe_ipmitool`, `agent_ipmitool` and `pxe_drac_inspector`, use ironic-inspector for their *inspect* interface.

The same functionality is available for all hardware types, but the appropriate `inspect` interface has to be enabled in the Bare Metal service configuration file, for example:

```
[DEFAULT]
enabled_inspect_interfaces = inspector,no-inspect
```

See *Enabling drivers and hardware types* for more details.

Note: The configuration option `[inspector]enabled` does not affect hardware types.

Then you can tell your nodes to use this interface, for example:

```
export OS_BAREMETAL_API_VERSION=1.31
for uuid in $(baremetal node list --driver ipmi -f value -c UUID); do
    baremetal node set <node> --inspect-interface inspector
done
```

Note: A node configured with the IPMI hardware type, will use the inspector inspection implementation automatically if it is enabled. This is not the case for the most of the vendor drivers.

Console

Several classic drivers, notably `pxe_ipmitool_socat` and `agent_ipmitool_socat`, use socat-based serial console implementation.

For the `ipmi` hardware type it is used by default, if enabled in the configuration file:

```
[DEFAULT]
enabled_console_interfaces = ipmitool-socat,no-console
```

If you want to use the `shellinabox` implementation instead, it has to be enabled as well:

```
[DEFAULT]
enabled_console_interfaces = ipmitool-shellinabox,no-console
```

Then you need to update some or all nodes to use it explicitly. For example, to update all nodes use:

```
export OS_BAREMETAL_API_VERSION=1.31
for uuid in $(baremetal node list --driver ipmi -f value -c UUID); do
    baremetal node set <node> --console-interface ipmitool-shellinabox
done
```

RAID

Many classic drivers, including `pxe_ipmitool` and `agent_ipmitool` use the IPA-based in-band RAID implementation by default.

For the hardware types it is not used by default. To use it, you need to enable it in the configuration first:

```
[DEFAULT]
enabled_raid_interfaces = agent,no-raid
```

Then you can update those nodes that support in-band RAID to use the `agent` RAID interface. For example, to update all nodes use:

```
export OS_BAREMETAL_API_VERSION=1.31
for uuid in $(baremetal node list --driver ipmi -f value -c UUID); do
    baremetal node set <node> --raid-interface agent
done
```

Note: The ability of a node to use the `agent` RAID interface depends on the ramdisk (more specifically, a hardware manager used in it), not on the driver.

Network and storage

The network and storage interfaces have always been dynamic, and thus do not require any special treatment during upgrade.

Vendor

Classic drivers are allowed to use the `VendorMixin` functionality to combine and expose several node or driver vendor passthru methods from different vendor interface implementations in one driver.

This is no longer possible with hardware types.

With hardware types, a vendor interface can only have a single active implementation from the list of vendor interfaces supported by a given hardware type.

Ironic no longer has in-tree drivers (both classic and hardware types) that rely on this `VendorMixin` functionality support. However if you are using an out-of-tree classic driver that depends on it, youll need to do the following in order to use vendor passthru methods from different vendor passthru implementations:

1. While creating a new hardware type to replace your classic driver, specify all vendor interface implementations your classic driver was using to build its `VendorMixin` as supported vendor interfaces (property `supported_vendor_interfaces` of the Python class that defines your hardware type).
2. Ensure all required vendor interfaces are enabled in the ironic configuration file under the `[DEFAULT]enabled_vendor_interfaces` option. You should also consider setting the `[DEFAULT]default_vendor_interface` option to specify the vendor interface for nodes that do not have one set explicitly.
3. Before invoking a specific vendor passthru method, make sure that the nodes vendor interface is set to the interface with the desired vendor passthru method. For example, if you want to invoke the vendor passthru method `vendor_method_foo()` from `vendor_foo` vendor interface:

```
# set the vendor interface to 'vendor_foo`
baremetal node set <node> --vendor-interface vendor_foo

# invoke the vendor passthru method
baremetal node passthru call <node> vendor_method_foo
```

CHAPTER

THREE

USER GUIDE

3.1 Bare Metal Service User Guide

Ironic is an OpenStack project which provisions bare metal (as opposed to virtual) machines. It may be used independently or as part of an OpenStack Cloud, and integrates with the OpenStack Identity (keystone), Compute (nova), Network (neutron), Image (glance) and Object (swift) services.

When the Bare Metal service is appropriately configured with the Compute and Network services, it is possible to provision both virtual and physical machines through the Compute services API. However, the set of instance actions is limited, arising from the different characteristics of physical servers and switch hardware. For example, live migration can not be performed on a bare metal instance.

The community maintains reference drivers that leverage open-source technologies (eg. PXE and IPMI) to cover a wide range of hardware. Ironics pluggable driver architecture also allows hardware vendors to write and contribute drivers that may improve performance or add functionality not provided by the community drivers.

3.1.1 Understanding Bare Metal service

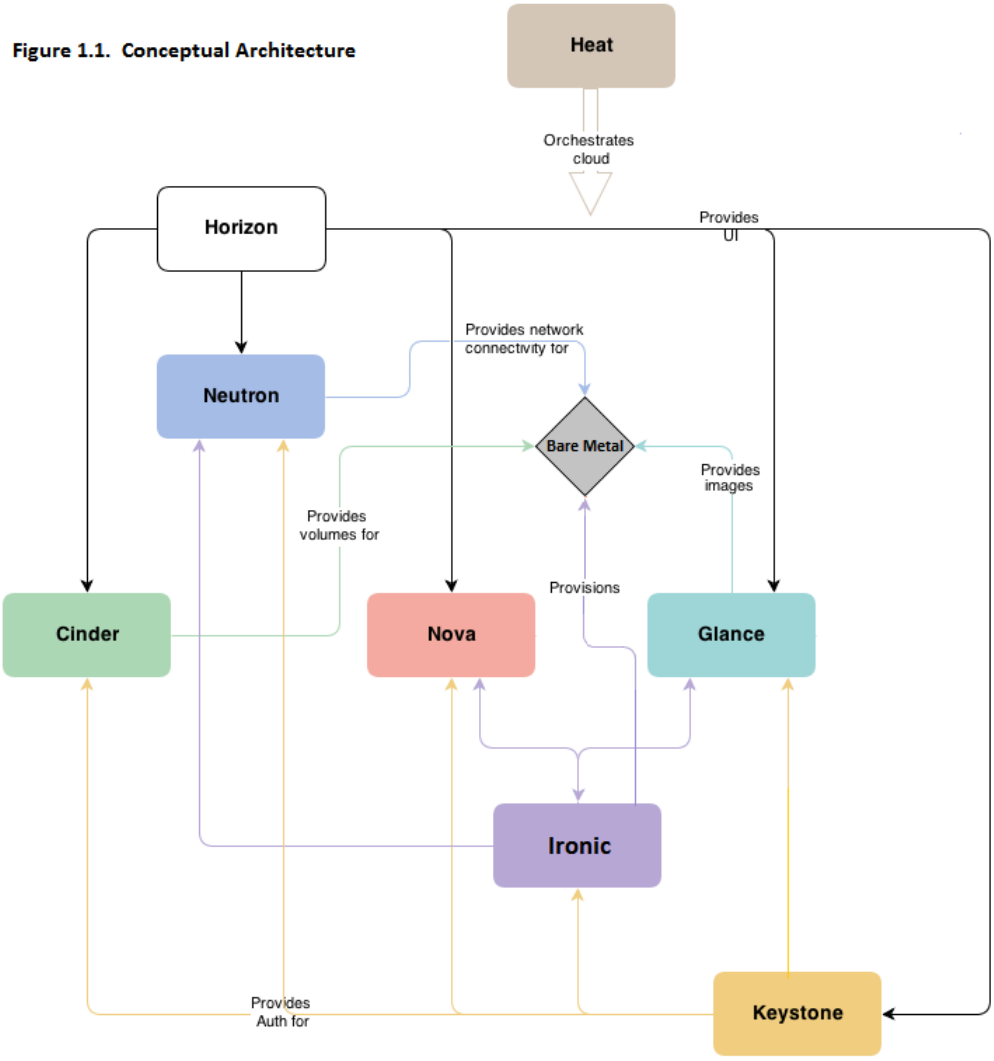
Why Provision Bare Metal

Here are a few use-cases for bare metal (physical server) provisioning in cloud; there are doubtless many more interesting ones:

- High-performance computing clusters
- Computing tasks that require access to hardware devices which cant be virtualized
- Database hosting (some databases run poorly in a hypervisor)
- Single tenant, dedicated hardware for performance, security, dependability and other regulatory requirements
- Or, rapidly deploying a cloud infrastructure

Conceptual Architecture

The following diagram shows the relationships and how all services come into play during the provisioning of a physical server. (Note that Ceilometer and Swift can be used with Ironic, but are missing from this diagram.)



Key Technologies for Bare Metal Hosting

Preboot Execution Environment (PXE)

PXE is part of the Wired for Management (WfM) specification developed by Intel and Microsoft. The PXE enables systems BIOS and network interface card (NIC) to bootstrap a computer from the network in place of a disk. Bootstrapping is the process by which a system loads the OS into local memory so that it can be executed by the processor. This capability of allowing a system to boot over a network simplifies server deployment and server management for administrators.

Dynamic Host Configuration Protocol (DHCP)

DHCP is a standardized networking protocol used on Internet Protocol (IP) networks for dynamically distributing network configuration parameters, such as IP addresses for interfaces and services. Using PXE, the BIOS uses DHCP to obtain an IP address for the network interface and to locate the server that stores the network bootstrap program (NBP).

Network Bootstrap Program (NBP)

NBP is equivalent to GRUB (GRand Unified Bootloader) or LILO (LInux LOader) - loaders which are traditionally used in local booting. Like the boot program in a hard drive environment, the NBP is responsible for loading the OS kernel into memory so that the OS can be bootstrapped over a network.

Trivial File Transfer Protocol (TFTP)

TFTP is a simple file transfer protocol that is generally used for automated transfer of configuration or boot files between machines in a local environment. In a PXE environment, TFTP is used to download NBP over the network using information from the DHCP server.

Intelligent Platform Management Interface (IPMI)

IPMI is a standardized computer system interface used by system administrators for out-of-band management of computer systems and monitoring of their operation. It is a method to manage systems that may be unresponsive or powered off by using only a network connection to the hardware rather than to an operating system.

Understanding Bare Metal Deployment

What happens when a boot instance request comes in? The below diagram walks through the steps involved during the provisioning of a bare metal instance.

These pre-requisites must be met before the deployment process:

- Dependent packages to be configured on the Bare Metal service node(s) where ironic-conductor is running like tftp-server, ipmi, syslinux etc for bare metal provisioning.
- Nova must be configured to make use of the bare metal service endpoint and compute driver should be configured to use ironic driver on the Nova compute node(s).
- Flavors to be created for the available hardware. Nova must know the flavor to boot from.
- Images to be made available in Glance. Listed below are some image types required for successful bare metal deployment:
  - bm-deploy-kernel
  - bm-deploy-ramdisk
  - user-image
  - user-image-vmlinuz
  - user-image-initrd

- Hardware to be enrolled via the bare metal API service.

Deploy Process

This describes a typical bare metal node deployment within OpenStack using PXE to boot the ramdisk. Depending on the ironic driver interfaces used, some of the steps might be marginally different, however the majority of them will remain the same.

1. A boot instance request comes in via the Nova API, through the message queue to the Nova scheduler.
2. Nova scheduler applies filters and finds the eligible hypervisor. The nova scheduler also uses the flavors `extra_specs`, such as `cpu_arch`, to match the target physical node.
3. Nova compute manager claims the resources of the selected hypervisor.
4. Nova compute manager creates (unbound) tenant virtual interfaces (VIFs) in the Networking service according to the network interfaces requested in the nova boot request. A caveat here is, the MACs of the ports are going to be randomly generated, and will be updated when the VIF is attached to some node to correspond to the node network interface cards (or bonds) MAC.
5. A spawn task is created by the nova compute which contains all the information such as which image to boot from etc. It invokes the `driver.spawn` from the virt layer of Nova compute. During the spawn process, the virt driver does the following:
    1. Updates the target ironic node with the information about deploy image, instance UUID, requested capabilities and various flavor properties.
    2. Validates nodes power and deploy interfaces, by calling the ironic API.
    3. Attaches the previously created VIFs to the node. Each neutron port can be attached to any ironic port or port group, with port groups having higher priority than ports. On ironic side, this work is done by the network interface. Attachment here means saving the VIF identifier into ironic port or port group and updating VIF MAC to match the ports or port groups MAC, as described in bullet point 4.
    4. Generates config drive, if requested.
6. Novas ironic virt driver issues a deploy request via the Ironic API to the Ironic conductor servicing the bare metal node.
7. Virtual interfaces are plugged in and Neutron API updates DHCP port to set PXE/TFTP options. In case of using `neutron` network interface, ironic creates separate provisioning ports in the Networking service, while in case of `flat` network interface, the ports created by nova are used both for provisioning and for deployed instance networking.
8. The ironic nodes boot interface prepares (i)PXE configuration and caches deploy kernel and ramdisk.
9. The ironic nodes management interface issues commands to enable network boot of a node.
10. The ironic nodes deploy interface caches the instance image, kernel and ramdisk if needed (it is needed in case of netboot for example).
11. The ironic nodes power interface instructs the node to power on.
12. The node boots the deploy ramdisk.

13. Depending on the exact driver used, the deploy ramdisk downloads the image from a URL (*Direct deploy*) or the conductor uses SSH to execute commands (*Ansible deploy*). The URL can be generated by Swift API-compatible object stores, for example Swift itself or RadosGW, or provided by a user.

    The image deployment is done.

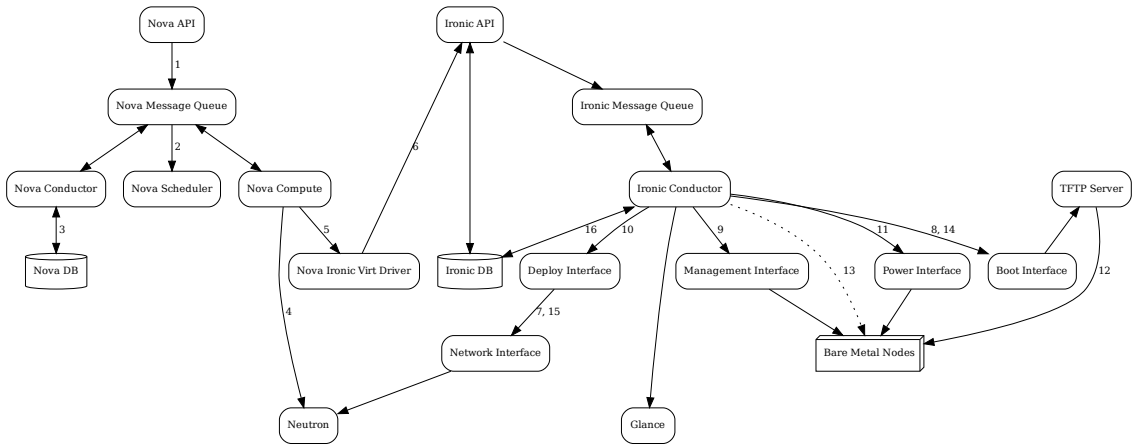
14. The nodes boot interface switches pxe config to refer to instance images (or, in case of local boot, sets boot device to disk), and asks the ramdisk agent to soft power off the node. If the soft power off by the ramdisk agent fails, the bare metal node is powered off via IPMI/BMC call.

15. The deploy interface triggers the network interface to remove provisioning ports if they were created, and binds the tenant ports to the node if not already bound. Then the node is powered on.

    Note: There are 2 power cycles during bare metal deployment; the first time the node is powered-on when ramdisk is booted, the second time after the image is deployed.

16. The bare metal nodes provisioning state is updated to `active`.

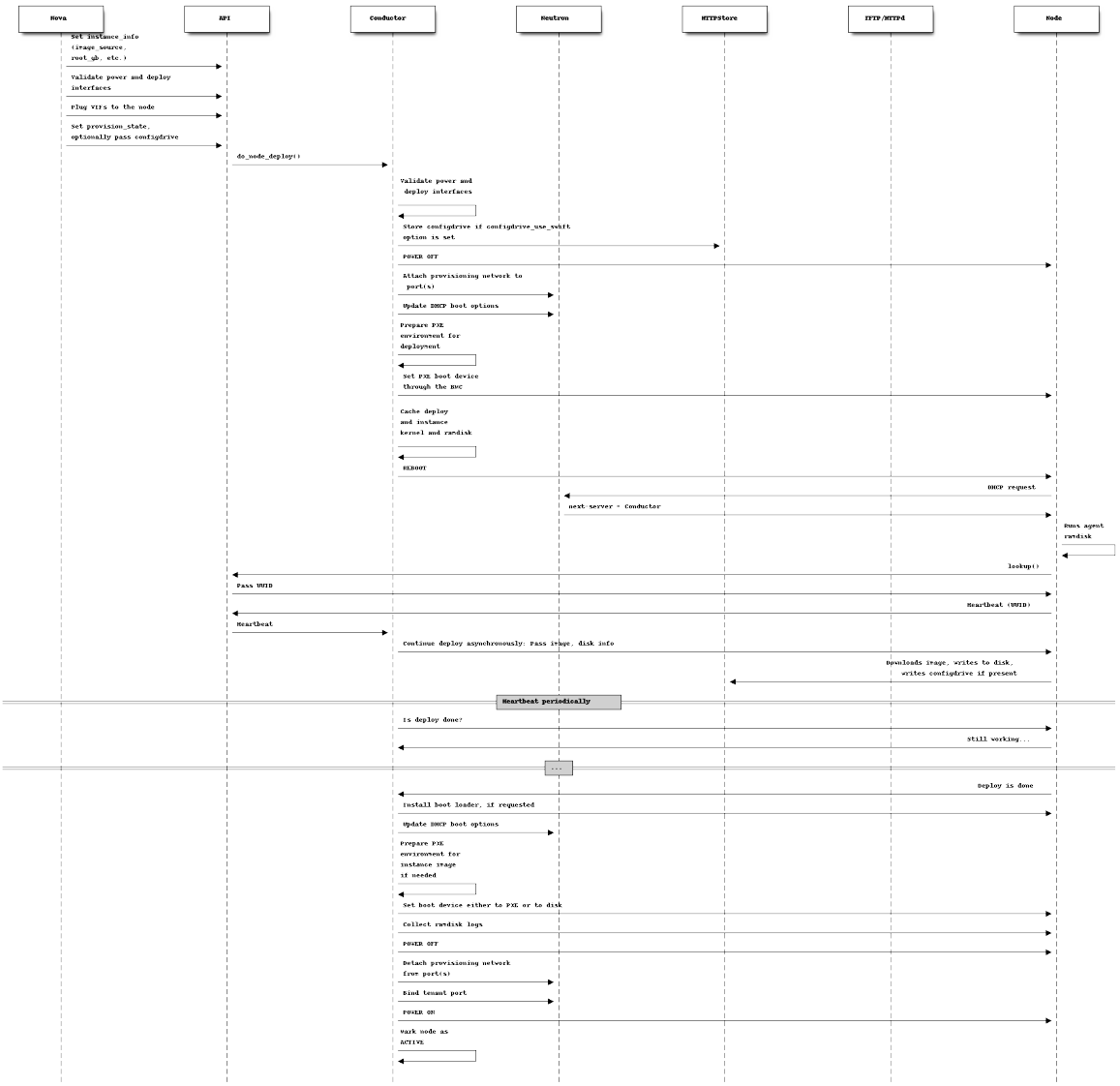
Below is the diagram that describes the above process.



The following two examples describe what ironic is doing in more detail, leaving out the actions performed by nova and some of the more advanced options.

Example: PXE Boot and Direct Deploy Process

This process is how *Direct deploy* works.



(From a talk and slides)

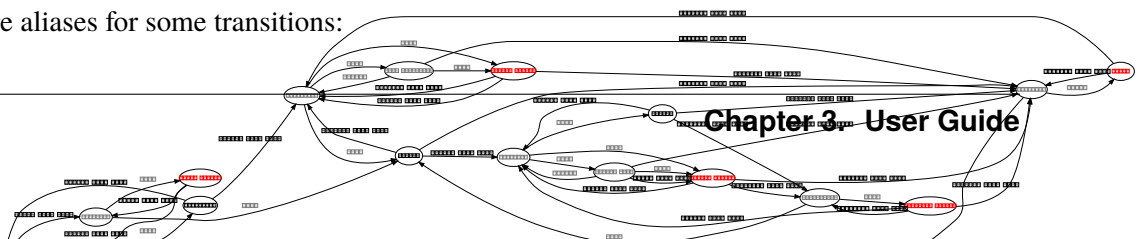
3.1.2 Bare Metal State Machine

State Machine Diagram

The diagram below shows the provisioning states that an Ironic node goes through during the lifetime of a node. The diagram also depicts the events that transition the node to different states.

Stable states are highlighted with a thicker border. All transitions from stable states are initiated by API requests. There are a few other API-initiated-transitions that are possible from non-stable states. The events for these API-initiated transitions are indicated with (via API). Internally, the conductor initiates the other transitions (depicted in gray).

Note: There are aliases for some transitions:



for `deleted`

Enrollment and Preparation

enroll (stable state) This is the state that all nodes start off in when created using API version 1.11 or newer. When a node is in the `enroll` state, the only thing ironic knows about it is that it exists, and ironic cannot take any further action by itself. Once a node has its driver/interfaces and their required information set in `node.driver_info`, the node can be transitioned to the `verifying` state by setting the nodes provision state using the `manage` verb.

See *Enrollment* for information on enrolling nodes.

verifying ironic will validate that it can manage the node using the information given in `node.driver_info` and with either the driver/hardware type and interfaces it has been assigned. This involves going out and confirming that the credentials work to access whatever node control mechanism they talk to.

manageable (stable state) Once ironic has verified that it can manage the node using the driver/interfaces and credentials passed in at node create time, the node will be transitioned to the `manageable` state. From `manageable`, nodes can transition to:

- `manageable` (through `cleaning`) by setting the nodes provision state using the `clean` verb.
- `manageable` (through `inspecting`) by setting the nodes provision state using the `inspect` verb.
- `available` (through `cleaning` if automatic cleaning is enabled) by setting the nodes provision state using the `provide` verb.
- `active` (through `adopting`) by setting the nodes provision state using the `adopt` verb.

`manageable` is the state that a node should be moved into when any updates need to be made to it such as changes to fields in driver_info and updates to networking information on ironic ports assigned to the node.

`manageable` is also the only stable state that can be transitioned to, from these failure states:

- `adopt failed`
- `clean failed`
- `inspect failed`

inspecting `inspecting` will utilize node introspection to update hardware-derived node properties to reflect the current state of the hardware. Typically, the node will transition to `manageable` if inspection is synchronous, or `inspect wait` if asynchronous. The node will transition to `inspect failed` if error occurred.

See *Hardware Inspection* for information about inspection.

inspect wait This is the provision state used when an asynchronous inspection is in progress. A successfully inspected node shall transition to `manageable` state.

inspect failed This is the state a node will move into when inspection of the node fails. From here the node can transitioned to:

- `inspecting` by setting the nodes provision state using the `inspect` verb.
- `manageable` by setting the nodes provision state using the `manage` verb

cleaning Nodes in the `cleaning` state are being scrubbed and reprogrammed into a known configuration.

When a node is in the `cleaning` state it means that the conductor is executing the clean step (for out-of-band clean steps) or preparing the environment (building PXE configuration files, configuring the DHCP, etc) to boot the ramdisk for running in-band clean steps.

clean wait Just like the `cleaning` state, the nodes in the `clean wait` state are being scrubbed and reprogrammed. The difference is that in the `clean wait` state the conductor is waiting for the ramdisk to boot or the clean step which is running in-band to finish.

The cleaning process of a node in the `clean wait` state can be interrupted by setting the nodes provision state using the `abort` verb if the task that is running allows it.

Deploy and Undeploy

available (stable state) After nodes have been successfully preconfigured and cleaned, they are moved into the `available` state and are ready to be provisioned. From `available`, nodes can transition to:

- `active` (through `deploying`) by setting the nodes provision state using the `active` or `deploy` verbs.
- `manageable` by setting the nodes provision state using the `manage` verb

deploying Nodes in `deploying` are being prepared to run a workload on them. This consists of running a series of tasks, such as:

- Setting appropriate BIOS configurations
- Partitioning drives and laying down file systems.
- Creating any additional resources (node-specific network config, a config drive partition, etc.) that may be required by additional subsystems.

See *Deploying with Bare Metal service* and *Node Deployment* for information about deploying nodes.

wait call-back Just like the `deploying` state, the nodes in `wait call-back` are being deployed. The difference is that in `wait call-back` the conductor is waiting for the ramdisk to boot or execute parts of the deployment which need to run in-band on the node (for example, installing the bootloader, or writing the image to the disk).

The deployment of a node in `wait call-back` can be interrupted by setting the nodes provision state using the `deleted` or `undeploy` verbs.

deploy failed This is the state a node will move into when a deployment fails, for example a timeout waiting for the ramdisk to PXE boot. From here the node can be transitioned to:

- `active` (through `deploying`) by setting the nodes provision state using the `active`, `deploy` or `rebuild` verbs.
- `available` (through `deleting` and `cleaning`) by setting the nodes provision state using the `deleted` or `undeploy` verbs.

active (stable state) Nodes in `active` have a workload running on them. ironic may collect out-of-band sensor information (including power state) on a regular basis. Nodes in `active` can transition to:

- `available` (through `deleting` and `cleaning`) by setting the nodes provision state using the `deleted` or `undeploy` verbs.
- `active` (through `deploying`) by setting the nodes provision state using the `rebuild` verb.
- `rescue` (through `rescuing`) by setting the nodes provision state using the `rescue` verb.

deleting Nodes in `deleting` state are being torn down from running an active workload. In `deleting`, ironic tears down and removes any configuration and resources it added in `deploying` or `rescuing`.

error (stable state) This is the state a node will move into when deleting an active deployment fails. From `error`, nodes can transition to:

- `available` (through `deleting` and `cleaning`) by setting the nodes provision state using the `deleted` or `undeploy` verbs.

adopting This state allows ironic to take over management of a baremetal node with an existing workload on it. Ordinarily when a baremetal node is enrolled and managed by ironic, it must transition through `cleaning` and `deploying` to reach `active` state. However, those baremetal nodes that have an existing workload on them, do not need to be deployed or cleaned again, so this transition allows these nodes to move directly from `manageable` to `active`.

See *Node adoption* for information about this feature.

Rescue

rescuing Nodes in `rescuing` are being prepared to perform rescue operations. This consists of running a series of tasks, such as:

- Setting appropriate BIOS configurations.
- Creating any additional resources (node-specific network config, etc.) that may be required by additional subsystems.

See *Rescue Mode* for information about this feature.

rescue wait Just like the `rescuing` state, the nodes in `rescue wait` are being rescued. The difference is that in `rescue wait` the conductor is waiting for the ramdisk to boot or execute parts of the rescue which need to run in-band on the node (for example, setting the password for user named `rescue`).

The rescue operation of a node in `rescue wait` can be aborted by setting the nodes provision state using the `abort` verb.

rescue failed This is the state a node will move into when a rescue operation fails, for example a timeout waiting for the ramdisk to PXE boot. From here the node can be transitioned to:

- `rescue` (through `rescuing`) by setting the nodes provision state using the `rescue` verb.
- `active` (through `unrescuing`) by setting the nodes provision state using the `unrescue` verb.
- `available` (through `deleting`) by setting the nodes provision state using the `deleted` verb.

rescue (stable state) Nodes in `rescue` have a rescue ramdisk running on them. Ironic may collect out-of-band sensor information (including power state) on a regular basis. Nodes in `rescue` can transition to:

- `active` (through `unrescuing`) by setting the nodes provision state using the `unrescue` verb.
- `available` (through `deleting`) by setting the nodes provision state using the `deleted` verb.

unrescuing Nodes in `unrescuing` are being prepared to transition to `active` state from `rescue` state. This consists of running a series of tasks, such as setting appropriate BIOS configurations such as changing boot device.

unrescue failed This is the state a node will move into when an unrescue operation fails. From here the node can be transitioned to:

- `rescue` (through `rescuing`) by setting the nodes provision state using the `rescue` verb.
- `active` (through `unrescuing`) by setting the nodes provision state using the `unrescue` verb.
- `available` (through `deleting`) by setting the nodes provision state using the `deleted` verb.

3.1.3 Creating instance images

Bare Metal provisioning requires two sets of images: the deploy images and the user images. The *deploy images* are used by the Bare Metal service to prepare the bare metal server for actual OS deployment. Whereas the user images are installed on the bare metal server to be used by the end user. There are two types of user images:

partition images contain only the contents of the root partition. Additionally, two more images are used together with them when booting from network: an image with a kernel and with an initramfs.

> **Warning:** To use partition images with local boot, Grub2 must be installed on them.

whole disk images contain a complete partition table with one or more partitions.

> **Warning:** The kernel/initramfs pair must not be used with whole disk images, otherwise theyll be mistaken for partition images.

Many distributions publish their own cloud images. These are usually whole disk images that are built for legacy boot mode (not UEFI), with Ubuntu being an exception (they publish images that work in both modes).

disk-image-builder

The disk-image-builder can be used to create user images required for deployment and the actual OS which the user is going to run.

- Install diskimage-builder package (use virtualenv, if you dont want to install anything globally):

```
# pip install diskimage-builder
```

- Build the image your users will run (Ubuntu image has been taken as an example):
  - Partition images

```
$ disk-image-create ubuntu baremetal dhcp-all-interfaces grub2 -o my-
↪image
```

  - Whole disk images

```
$ disk-image-create ubuntu vm dhcp-all-interfaces -o my-image
```

    with an EFI partition:

```
$ disk-image-create ubuntu vm block-device-efi dhcp-all-interfaces -
↪o my-image
```

The partition image command creates `my-image.qcow2`, `my-image.vmlinuz` and `my-image.initrd` files. The `grub2` element in the partition image creation command is only needed if local boot will be used to deploy `my-image.qcow2`, otherwise the images `my-image.vmlinuz` and `my-image.initrd` will be used for PXE booting after deploying the bare metal with `my-image.qcow2`. For whole disk images only the main image is used.

If you want to use Fedora image, replace `ubuntu` with `fedora` in the chosen command.

Virtual machine

Virtual machine software can also be used to build user images. There are different software options available, qemu-kvm is usually a good choice on linux platform, it supports emulating many devices and even building images for architectures other than the host machine by software emulation. VirtualBox is another good choice for non-linux host.

The procedure varies depending on the software used, but the steps for building an image are similar, the user creates a virtual machine, and installs the target system just like what is done for a real hardware. The system can be highly customized like partition layout, drivers or software shipped, etc.

Usually libvirt and its management tools are used to make interaction with qemu-kvm easier, for example, to create a virtual machine with `virt-install`:

```
$ virt-install --name centos8 --ram 4096 --vcpus=2 -f centos8.qcow2 \
> --cdrom CentOS-8-x86_64-1905-dvd1.iso
```

Graphic frontend like `virt-manager` can also be utilized.

The disk file can be used as user image after the system is set up and powered off. The path of the disk file varies depending on the software used, usually its stored in a user-selected part of the local file system. For qemu-kvm or GUI frontend building upon it, its typically stored at `/var/lib/libvirt/images`.

3.1.4 Deploying with Bare Metal service

This guide explains how to use Ironic to deploy nodes without any front-end service, such as OpenStack Compute (nova) or Metal3.

Note: To simplify this task you can use the metalsmith tool which provides a convenient CLI for the most common cases.

Allocations

Allocation is a way to find and reserve a node suitable for deployment. When an allocation is created, the list of available nodes is searched for a node with the given *resource class* and *traits*, similarly to how it is done in *OpenStack Compute flavors*. Only the resource class is mandatory, for example:

```
$ baremetal allocation create --resource-class baremetal --wait
+-----------------+--------------------------------------+
| Field           | Value                                |
+-----------------+--------------------------------------+
| candidate_nodes | []                                   |
| created_at      | 2019-04-03T12:18:26+00:00            |
| extra           | {}                                   |
| last_error      | None                                 |
| name            | None                                 |
| node_uuid       | 5d946337-b1d9-4b06-8eda-4fb77e994a0d |
| resource_class  | baremetal                            |
| state           | active                               |
| traits          | []                                   |
| updated_at      | 2019-04-03T12:18:26+00:00            |
| uuid            | e84f5d60-84f1-4701-a635-10ff90e2f3b0 |
+-----------------+--------------------------------------+
```

Note: The allocation processing is fast but nonetheless asynchronous. Use the `--wait` argument to wait for the results.

If an allocation is successful, it sets the nodes `instance_uuid` to the allocation UUID. The nodes UUID can be retrieved from the allocations `node_uuid` field.

An allocation is automatically deleted when the associated node is unprovisioned. If you dont provision the node, youre responsible for deleting the allocation.

See the allocation API reference for more information on how to use allocations.

Populating instance information

The nodes `instance_info` field is a JSON object that contains all information required for deploying an instance on bare metal. It has to be populated before deployment and is automatically cleared on tear down.

Image information

You need to specify image information in the nodes `instance_info` (see *Creating instance images*):

- `image_source` - URL of the whole disk or root partition image, mandatory. The following schemes are supported: `http://`, `https://` and `file://`. Files have to be accessible by the conductor. If the scheme is missing, an Image Service (glance) image UUID is assumed.
- `root_gb` - size of the root partition, required for partition images.

  Note: Older versions of the Bare Metal service used to require a positive integer for `root_gb` even for whole-disk images. You may want to set it for compatibility.

- `image_checksum` - MD5 checksum of the image specified by `image_source`, only required for `http://` images when using *Direct deploy*.

  Other checksum algorithms are supported via the `image_os_hash_algo` and `image_os_hash_value` fields. They may be used instead of the `image_checksum` field.

  Warning: If your operating system is running in FIPS 140-2 mode, MD5 will not be available, and you **must** use SHA256 or another modern algorithm.

  Starting with the Stein release of ironic-python-agent can also be a URL to a checksums file, e.g. one generated with:

  ```
  $ cd /path/to/http/root
  $ md5sum *.img > checksums
  ```

- `kernel`, `ramdisk` - HTTP(s) or file URLs of the kernel and initramfs of the target OS. Must be added **only** for partition images and only if network boot is required. Supports the same schemes as `image_source`.

An example for a partition image with local boot:

```
baremetal node set $NODE_UUID \
    --instance-info image_source=http://image.server/my-image.qcow2 \
    --instance-info image_checksum=1f9c0e1bad977a954ba40928c1e11f33 \
    --instance-info image_type=partition \
    --instance-info root_gb=10
```

With a SHA256 hash:

```
baremetal node set $NODE_UUID \
    --instance-info image_source=http://image.server/my-image.qcow2 \
```

(continues on next page)

(continued from previous page)

```
    --instance-info image_os_hash_algo=sha256 \
    --instance-info image_os_hash_
↪value=a64dd95e0c48e61ed741ff026d8c89ca38a51f3799955097c5123b1705ef13d4 \
    --instance-info image_type=partition \
    --instance-info root_gb=10
```

If you use network boot (or Ironic before Yoga), two more fields must be set:

```
baremetal node set $NODE_UUID \
    --instance-info image_source=http://image.server/my-image.qcow2 \
    --instance-info image_checksum=1f9c0e1bad977a954ba40928c1e11f33 \
    --instance-info image_type=partition \
    --instance-info kernel=http://image.server/my-image.kernel \
    --instance-info ramdisk=http://image.server/my-image.initramfs \
    --instance-info root_gb=10
```

With a whole disk image and a checksum URL:

```
baremetal node set $NODE_UUID \
    --instance-info image_source=http://image.server/my-image.qcow2 \
    --instance-info image_checksum=http://image.server/my-image.qcow2.CHECKSUM
```

Note: Certain hardware types and interfaces may require additional or different fields to be provided. See specific guides under *Drivers, Hardware Types and Hardware Interfaces*.

When using low RAM nodes with `http://` images that are not in the RAW format, you may want them cached locally, converted to raw and served from the conductors HTTP server:

```
baremetal node set $NODE_UUID --instance-info image_download_source=local
```

For software RAID with whole-disk images, the root UUID of the root partition has to be provided so that the bootloader can be correctly installed:

```
baremetal node set $NODE_UUID --instance-info image_rootfs_uuid=<uuid>
```

Capabilities

- *Boot mode* can be specified per instance:

  ```
  baremetal node set $NODE_UUID \
      --instance-info capabilities='{"boot_mode": "uefi"}'
  ```

  Otherwise, the `boot_mode` capability from the nodes `properties` will be used.

  Warning: The two settings must not contradict each other.

Note: This capability was introduced in the Wallaby release series, previously ironic used a separate `instance_info/deploy_boot_mode` field instead.

- To override the *boot option* used for this instance, set the `boot_option` capability:

```
baremetal node set $NODE_UUID \
    --instance-info capabilities='{"boot_option": "local"}'
```

- Starting with the Ussuri release, you can set *root device hints* per instance:

```
baremetal node set $NODE_UUID \
    --instance-info root_device='{"wwn": "0x4000cca77fc4dba1"}'
```

This setting overrides any previous setting in `properties` and will be removed on undeployment.

Overriding a hardware interface

Non-admins with temporary access to a node, may wish to specify different node interfaces. However, allowing them to set these interface values directly on the node is problematic, as there is no automated way to ensure that the original interface values are restored.

In order to temporarily override a hardware interface, simply set the appropriate value in `instance_info`. For example, if youd like to override a nodes storage interface, run the following:

```
baremetal node set $NODE_UUID --instance-info storage_interface=cinder
```

`instance_info` values persist until after a node is cleaned.

Note: This feature is available starting with the Wallaby release.

Attaching virtual interfaces

If using the OpenStack Networking service (neutron), you can attach its ports to a node before deployment as VIFs:

```
baremetal node vif attach $NODE_UUID $PORT_UUID
```

Warning: These are **neutron** ports, not **ironic** ports!

VIFs are automatically detached on deprovisioning.

Deployment

1. Validate that all parameters are correct:

```
$ baremetal node validate $NODE_UUID
+------------+--------+----------------------------------------------------
↪-------------+
| Interface  | Result | Reason                                             ␣
↪             |
+------------+--------+----------------------------------------------------
↪-------------+
| boot       | True   |                                                    ␣
↪             |
| console    | False  | Missing 'ipmi_terminal_port' parameter in node's␣
↪driver_info.  |
| deploy     | True   |                                                    ␣
↪             |
| inspect    | True   |                                                    ␣
↪             |
| management | True   |                                                    ␣
↪             |
| network    | True   |                                                    ␣
↪             |
| power      | True   |                                                    ␣
↪             |
| raid       | True   |                                                    ␣
↪             |
| storage    | True   |                                                    ␣
↪             |
+------------+--------+----------------------------------------------------
↪-------------+
```

2. Now you can start the deployment, run:

```
baremetal node deploy $NODE_UUID
```

3. Starting with the Wallaby release you can also request custom deploy steps, see *Requesting steps* for details.

Deploying with a config drive

The configuration drive is a small image used to store instance-specific metadata and is present to the instance as a disk partition labeled `config-2`. See *Enabling the configuration drive (configdrive)* for a detailed explanation.

A configuration drive can be provided either as a whole ISO 9660 image or as JSON input for building an image. A first-boot service, such as cloud-init, must be running on the instance image for the configuration to be applied.

Building a config drive on the client side

For the format of the configuration drive, Bare Metal service expects a `gzipped` and `base64` encoded ISO 9660 file with a `config-2` label. The baremetal client can generate a configuration drive in the expected format. Pass a directory path containing the files that will be injected into it via the `--config-drive` parameter of the `baremetal node deploy` command, for example:

```
baremetal node deploy $NODE_UUID --config-drive /dir/configdrive_files
```

Note: A configuration drive could also be a data block with a VFAT filesystem on it instead of ISO 9660. But its unlikely that it would be needed since ISO 9660 is widely supported across operating systems.

Building a config drive on the conductor side

Starting with the Stein release and *ironicclient* 2.7.0, you can request building a configdrive on the server side by providing a JSON with keys `meta_data`, `user_data` and `network_data` (all optional), e.g.:

```
baremetal node deploy $node_identifier \
    --config-drive '{"meta_data": {"hostname": "server1.cluster"}}'
```

Note: When this feature is used, host name defaults to the nodes name or UUID.

SSH public keys can be provided as a mapping:

```
baremetal node deploy $NODE_UUID \
    --config-drive '{"meta_data": {"public_keys": {"0": "ssh key contents"}}}'
```

If using cloud-init, its configuration can be supplied as `user_data`, e.g.:

```
baremetal node deploy $NODE_UUID \
    --config-drive '{"user_data": "#cloud-config\n{\"users\": [{\"name\": ...}
↪]}"}'
```

Warning: User data is a string, not a JSON! Also note that a prefix, such as `#cloud-config`, is required, see user data format.

Some first-boot services support network configuration in the OpenStack network data format. It can be provided in the `network_data` field of the configuration drive.

Ramdisk booting

Advanced operators, specifically ones working with ephemeral workloads, may find it more useful to explicitly treat a node as one that would always boot from a Ramdisk. See *Booting a Ramdisk or an ISO* for details.

CHAPTER

FOUR

ADMINISTRATOR GUIDE

4.1 Drivers, Hardware Types and Hardware Interfaces

4.1.1 Generic Interfaces

Boot interfaces

The boot interface manages booting of both the deploy ramdisk and the user instances on the bare metal node.

The *PXE boot* interface is generic and works with all hardware that supports booting from network. Alternatively, several vendors provide *virtual media* implementations of the boot interface. They work by pushing an ISO image to the nodes management controller, and do not require either PXE or iPXE. Check your driver documentation at *Drivers, Hardware Types and Hardware Interfaces* for details.

PXE boot

The `pxe` boot interface uses PXE or iPXE to deliver the target kernel/ramdisk pair. PXE uses relatively slow and unreliable TFTP protocol for transfer, while iPXE uses HTTP. The downside of iPXE is that its less common, and usually requires bootstrapping using PXE first.

The `pxe` boot interface works by preparing a PXE/iPXE environment for a node on the file system, then instructing the DHCP provider (for example, the Networking service) to boot the node from it. See ref:*direct-deploy-example* for a better understanding of the whole deployment process.

Note: Both PXE and iPXE are configured differently, when UEFI boot is used instead of conventional BIOS boot. This is particularly important for CPU architectures that do not have BIOS support at all.

The `pxe` boot interface is used by default for many hardware types, including `ipmi`. Some hardware types, notably `ilo` and `irmc` have their specific implementations of the PXE boot interface.

Additional configuration is required for this boot interface - see *Configuring PXE and iPXE* for details.

Common options

Enable persistent boot device for deploy/clean operation

For (i)PXE booting, Ironic uses non-persistent boot order changes for clean/deploy by default. For some drivers, persistent changes are far more costly than non-persisent ones, so this approach can bring a performance benefit.

In order to control this behavior, however, Ironic provides the `force_persistent_boot_device` flag in the nodes `driver_info`. It allows the values `Default` (make all changes but the last one upon deployment non-persistent), `Always` (make all changes persistent), and `Never` (make all boot order changes non-persistent). For example in order to have only persistent changes one would need to set something like:

```
$ openstack baremetal node set --driver-info force_persistent_boot_device=
↪'Always' <node>
```

Note: It is recommended to check if the nodes state has not changed as there is no way of locking the node between these commands.

Note: The values True/False for the option force_persistent_boot_device in the nodes driver info for the (i)PXE drivers are deprecated and support for them may be removed in a future release. The former default value False is replaced by the new value Default, the value True is replaced by Always.

Deploy Interfaces

A *deploy* interface plays a critical role in the provisioning process. It orchestrates the whole deployment and defines how the image gets transferred to the target disk.

Direct deploy

With `direct` deploy interface, the deploy ramdisk fetches the image from an HTTP location. It can be an object storage (swift or RadosGW) temporary URL or a user-provided HTTP URL. The deploy ramdisk then copies the image to the target disk. See *direct deploy diagram* for a detailed explanation of how this deploy interface works.

You can specify this deploy interface when creating or updating a node:

```
baremetal node create --driver ipmi --deploy-interface direct
baremetal node set <NODE> --deploy-interface direct
```

Note: For historical reasons the `direct` deploy interface is sometimes called `agent`. This is because before the Kilo release **ironic-python-agent** used to only support this deploy interface.

Deploy with custom HTTP servers

The `direct` deploy interface can also be configured to use with custom HTTP servers set up at ironic conductor nodes, images will be cached locally and made accessible by the HTTP server.

To use this deploy interface with a custom HTTP server, set `image_download_source` to `http` in the `[agent]` section.

```
[agent]
...
image_download_source = http
...
```

This configuration affects *glance* and `file://` images. If you want `http(s)://` images to also be cached and served locally, use instead:

```
[agent]
image_download_source = local
```

Note: This option can also be set per node in `driver_info`:

```
baremetal node set <node> --driver-info image_download_source=local
```

or per instance in `instance_info`:

```
baremetal node set <node> --instance-info image_download_source=local
```

You need to set up a workable HTTP server at each conductor node which with `direct` deploy interface enabled, and check http related options in the ironic configuration file to match the HTTP server configurations.

```
[deploy]
http_url = http://example.com
http_root = /httpboot
```

Note: See also: *Deploying outside of the provisioning network*.

Each HTTP server should be configured to follow symlinks for images accessible from HTTP service. Please refer to configuration option `FollowSymLinks` if you are using Apache HTTP server, or `disable_symlinks` if Nginx HTTP server is in use.

Ansible deploy

This interface is similar to `direct` in the sense that the image is downloaded by the ramdisk directly from the image store (not from ironic-conductor host), but the logic of provisioning the node is held in a set of Ansible playbooks that are applied by the `ironic-conductor` service handling the node. While somewhat more complex to set up, this deploy interface provides greater flexibility in terms of advanced node preparation during provisioning.

This interface is supported by most but not all hardware types declared in ironic. However this deploy interface is not enabled by default. To enable it, add `ansible` to the list of enabled deploy interfaces in `enabled_deploy_interfaces` option in the `[DEFAULT]` section of ironics configuration file:

```
[DEFAULT]
...
enabled_deploy_interfaces = direct,ansible
...
```

Once enabled, you can specify this deploy interface when creating or updating a node:

```
baremetal node create --driver ipmi --deploy-interface ansible
baremetal node set <NODE> --deploy-interface ansible
```

For more information about this deploy interface, its features and how to use it, see *Ansible deploy interface*.

Ansible deploy interface

Ansible is a mature and popular automation tool, written in Python and requiring no agents running on the node being configured. All communications with the node are by default performed over secure SSH transport.

The `ansible` deploy interface uses Ansible playbooks to define the deployment logic. It is not based on Ironic Python Agent (IPA) and does not generally need IPA to be running in the deploy ramdisk.

Overview

The main advantage of this deploy interface is extended flexibility in regards to changing and adapting node deployment logic for specific use cases, via Ansible tooling that is already familiar to operators.

It can be used to shorten the usual feature development cycle of

- implementing logic in ironic,
- implementing logic in IPA,
- rebuilding deploy ramdisk,
- uploading deploy ramdisk to Glance/HTTP storage,
- reassigning deploy ramdisk to nodes,
- restarting ironic-conductor service(s) and
- running a test deployment

by using a stable deploy ramdisk and not requiring ironic-conductor restarts (see *Extending playbooks*).

The main disadvantage of this deploy interface is the synchronous manner of performing deployment/cleaning tasks. A separate `ansible-playbook` process is spawned for each node being provisioned or cleaned, which consumes one thread from the thread pool available to the `ironic-conductor` process and blocks this thread until the node provisioning or cleaning step is finished or fails. This has to be taken into account when planning an ironic deployment that enables this deploy interface.

Each action (deploy, clean) is described by a single playbook with roles, which is run whole during deployment, or tag-wise during cleaning. Control of cleaning steps is through tags and auxiliary clean steps file. The playbooks for actions can be set per-node, as can the clean steps file.

Features

Similar to deploy interfaces relying on Ironic Python Agent (IPA), this deploy interface also depends on the deploy ramdisk calling back to ironic APIs `heartbeat` endpoint.

However, the driver is currently synchronous, so only the first heartbeat is processed and is used as a signal to start `ansible-playbook` process.

User images

Supports whole-disk images and partition images:

- compressed images are downloaded to RAM and converted to disk device;
- raw images are streamed to disk directly.

For partition images the driver will create root partition, and, if requested, ephemeral and swap partitions as set in nodes `instance_info` by the Compute service or operator. The create partition table will be of `msdos` type by default, the nodes `disk_label` capability is honored if set in nodes `instance_info` (see also *Choosing the disk label*).

Configdrive partition

Creating a configdrive partition is supported for both whole disk and partition images, on both `msdos` and `GPT` labeled disks.

Root device hints

Root device hints are currently supported in their basic form only, with exact matches (see *Specifying the disk for deployment (root device hints)* for more details). If no root device hint is provided for the node, the first device returned as part of `ansible_devices` fact is used as root device to create partitions on or write the whole disk image to.

Node cleaning

Cleaning is supported, both automated and manual. The driver has two default clean steps:

- wiping device metadata
- disk shredding

Their priority can be overridden via `[deploy]\erase_devices_metadata_priority` and `[deploy]\erase_devices_priority` options, respectively, in the ironic configuration file.

As in the case of this driver all cleaning steps are known to the ironic-conductor service, booting the deploy ramdisk is completely skipped when there are no cleaning steps to perform.

Note: Aborting cleaning steps is not supported.

Logging

Logging is implemented as custom Ansible callback module, that makes use of `oslo.log` and `oslo.config` libraries and can re-use logging configuration defined in the main ironic configuration file to set logging for Ansible events, or use a separate file for this purpose.

It works best when `journald` support for logging is enabled.

Requirements

Ansible Tested with, and targets, Ansible 2.5.x

Bootstrap image requirements

- password-less sudo permissions for the user used by Ansible
- python 2.7.x
- openssh-server
- GNU coreutils
- utils-linux
- parted
- gdisk
- qemu-utils
- python-requests (for ironic callback and streaming image download)
- python-netifaces (for ironic callback)

A set of scripts to build a suitable deploy ramdisk based on TinyCore Linux and `tinyipa` ramdisk, and an element for `diskimage-builder` can be found in ironic-staging-drivers project but will be eventually migrated to the new ironic-python-agent-builder project.

Setting up your environment

1. Install ironic (either as part of OpenStack or standalone)
   - If using ironic as part of OpenStack, ensure that the Image service is configured to use the Object Storage service as backend, and the Bare Metal service is configured accordingly, see *Configure the Image service for temporary URLs*.
2. Install Ansible version as specified in `ironic/driver-requirements.txt` file
3. Edit ironic configuration file
   A. Add `ansible` to the list of deploy interfaces defined in `[DEFAULT]\ enabled_deploy_interfaces` option.
   B. Ensure that a hardware type supporting `ansible` deploy interface is enabled in `[DEFAULT]\ enabled_hardware_types` option.
   C. Modify options in the `[ansible]` section of ironics configuration file if needed (see *Configuration file*).
4. (Re)start ironic-conductor service
5. Build suitable deploy kernel and ramdisk images
6. Upload them to Glance or put in your HTTP storage
7. Create new or update existing nodes to use the enabled driver of your choice and populate *Driver properties for the Node* when different from defaults.
8. Deploy the node as usual.

Ansible-deploy options

Configuration file

Driver options are configured in `[ansible]` section of ironic configuration file, for their descriptions and default values please see configuration file sample.

Driver properties for the Node

Set them per-node via `baremetal node set` command, for example:

```
baremetal node set <node> \
    --deploy-interface ansible \
    --driver-info ansible_username=stack \
    --driver-info ansible_key_file=/etc/ironic/id_rsa
```

ansible_username User name to use for Ansible to access the node. Default is taken from `[ansible]/ default_username` option of the ironic configuration file (defaults to `ansible`).

ansible_key_file Private SSH key used to access the node. Default is taken from `[ansible]/ default_key_file` option of the ironic configuration file. If neither is set, the default private SSH keys of the user running the `ironic-conductor` process will be used.

ansible_deploy_playbook Playbook to use when deploying this node. Default is taken from `[ansible]/default_deploy_playbook` option of the ironic configuration file (defaults to `deploy.yaml`).

ansible_shutdown_playbook Playbook to use to gracefully shutdown the node in-band. Default is taken from `[ansible]/default_shutdown_playbook` option of the ironic configuration file (defaults to `shutdown.yaml`).

ansible_clean_playbook Playbook to use when cleaning the node. Default is taken from `[ansible]/default_clean_playbook` option of the ironic configuration file (defaults to `clean.yaml`).

ansible_clean_steps_config Auxiliary YAML file that holds description of cleaning steps used by this node, and defines playbook tags in `ansible_clean_playbook` file corresponding to each cleaning step. Default is taken from `[ansible]/default_clean_steps_config` option of the ironic configuration file (defaults to `clean_steps.yaml`).

ansible_python_interpreter Absolute path to the python interpreter on the managed machine. Default is taken from `[ansible]/default_python_interpreter` option of the ironic configuration file. Ansible uses `/usr/bin/python` by default.

Customizing the deployment logic

Expected playbooks directory layout

The `[ansible]\playbooks_path` option in the ironic configuration file is expected to have a standard layout for an Ansible project with some additions:

```
<playbooks_path>
|
\_ inventory
\_ add-ironic-nodes.yaml
\_ roles
 \_ role1
 \_ role2
 \_ ...
|
\_callback_plugins
 \_ ...
|
\_ library
 \_ ...
```

The extra files relied by this driver are:

inventory Ansible inventory file containing a single entry of `conductor ansible_connection=local`. This basically defines an alias to `localhost`. Its purpose is to make logging for tasks performed by Ansible locally and referencing the localhost in playbooks more intuitive. This also suppresses warnings produced by Ansible about `hosts` file being empty.

add-ironic-nodes.yaml This file contains an Ansible play that populates in-memory Ansible inventory with access information received from the ansible-deploy interface, as well as some per-node variables. Include it in all your custom playbooks as the first play.

The default `deploy.yaml` playbook is using several smaller roles that correspond to particular stages of deployment process:

- `discover` - e.g. set root device and image target
- `prepare` - if needed, prepare system, for example create partitions
- `deploy` - download/convert/write user image and configdrive
- `configure` - post-deployment steps, e.g. installing the bootloader

Some more included roles are:

- `shutdown` - used to gracefully power the node off in-band
- `clean` - defines cleaning procedure, with each clean step defined as separate playbook tag.

Extending playbooks

Most probably youd start experimenting like this:

1. Create a copy of `deploy.yaml` playbook *in the same folder*, name it distinctively.
2. Create Ansible roles with your customized logic in `roles` folder.
   A. In your custom deploy playbook, replace the `prepare` role with your own one that defines steps to be run *before* image download/writing. This is a good place to set facts overriding those provided/omitted by the driver, like `ironic_partitions` or `ironic_root_device`, and create custom partitions or (software) RAIDs.
   B. In your custom deploy playbook, replace the `configure` role with your own one that defines steps to be run *after* image is written to disk. This is a good place for example to configure the bootloader and add kernel options to avoid additional reboots.
   C. Use those new roles in your new playbook.
3. Assign the custom deploy playbook youve created to the nodes `driver_info/ansible_deploy_playbook` field.
4. Run deployment.
   A. No ironic-conductor restart is necessary.
   B. A new deploy ramdisk must be built and assigned to nodes only when you want to use a command/script/package not present in the current deploy ramdisk and you can not or do not want to install those at runtime.

Variables you have access to

This driver will pass the single JSON-ified extra var argument to Ansible (as in `ansible-playbook -e ..`). Those values are then accessible in your plays as well (some of them are optional and might not be defined):

```
ironic:
  nodes:
  - ip: "<IPADDRESS>"
    name: "<NODE_UUID>"
```

(continues on next page)

(continued from previous page)

```
    user: "<USER ANSIBLE WILL USE>"
    extra: "<COPY OF NODE's EXTRA FIELD>"
  image:
    url: "<URL TO FETCH THE USER IMAGE FROM>"
    disk_format: "<qcow2|raw|...>"
    container_format: "<bare|...>"
    checksum: "<hash-algo:hashstring>"
    mem_req: "<REQUIRED FREE MEMORY TO DOWNLOAD IMAGE TO RAM>"
    tags: "<LIST OF IMAGE TAGS AS DEFINED IN GLANCE>"
    properties: "<DICT OF IMAGE PROPERTIES AS DEFINED IN GLANCE>"
  configdrive:
    type: "<url|file>"
    location: "<URL OR PATH ON CONDUCTOR>"
  partition_info:
    label: "<msdos|gpt>"
    preserve_ephemeral: "<bool>"
    ephemeral_format: "<FILESYSTEM TO CREATE ON EPHEMERAL PARTITION>"
    partitions: "<LIST OF PARTITIONS IN FORMAT EXPECTED BY PARTED MODULE>"
  raid_config: "<COPY OF NODE's TARGET_RAID_CONFIG FIELD>"
```

ironic.nodes List of dictionaries (currently of only one element) that will be used by `add-ironic-nodes.yaml` play to populate in-memory inventory. It also contains a copy of nodes `extra` field so you can access it in the playbooks. The Ansibles host is set to nodes UUID.

ironic.image All fields of nodes `instance_info` that start with `image_` are passed inside this variable. Some extra notes and fields:

- `mem_req` is calculated from image size (if available) and config option `[ansible]extra_memory`.
- if `checksum` is not in the form `<hash-algo>:<hash-sum>`, hashing algorithm is assumed to be `md5` (default in Glance).
- `validate_certs` - boolean (`yes`/`no`) flag that turns validating image store SSL certificate on or off (default is yes). Governed by `[ansible]image_store_insecure` option in ironic configuration file.
- `cafile` - custom CA bundle to use for validating image store SSL certificate. Takes value of `[ansible]image_store_cafile` if that is defined. Currently is not used by default playbooks, as Ansible has no way to specify the custom CA bundle to use for single HTTPS actions, however you can use this value in your custom playbooks to for example upload and register this CA in the ramdisk at deploy time.
- `client_cert` - cert file for client-side SSL authentication. Takes value of `[ansible]image_store_certfile` option if defined. Currently is not used by default playbooks, however you can use this value in your custom playbooks.
- `client_key` - private key file for client-side SSL authentication. Takes value of `[ansible]image_store_keyfile` option if defined. Currently is not used by default playbooks, however you can use this value in your custom playbooks.

ironic.partition_info.partitions Optional. List of dictionaries defining partitions to create on the node in the form:

```
partitions:
- name: "<NAME OF PARTITION>"
  unit: "<UNITS FOR SIZE>"
  size: "<SIZE OF THE PARTITION>"
  type: "<primary|extended|logical>"
  align: "<ONE OF PARTED_SUPPORTED OPTIONS>"
  format: "<PARTITION TYPE TO SET>"
  flags:
    flag_name: "<bool>"
```

The driver will populate this list from `root_gb`, `swap_mb` and `ephemeral_gb` fields of `instance_info`. The driver will also prepend the `bios_grub`-labeled partition when deploying on GPT-labeled disk, and pre-create a 64 MiB partition for configdrive if it is set in `instance_info`.

Please read the documentation included in the `ironic_parted` modules source for more info on the module and its arguments.

`ironic.partition_info.ephemeral_format` Optional. Taken from `instance_info`, it defines file system to be created on the ephemeral partition. Defaults to the value of `[pxe]\ default_ephemeral_format` option in ironic configuration file.

`ironic.partition_info.preserve_ephemeral` Optional. Taken from the `instance_info`, it specifies if the ephemeral partition must be preserved or rebuilt. Defaults to `no`.

`ironic.raid_config` Taken from the `target_raid_config` if not empty, it specifies the RAID configuration to apply.

As usual for Ansible playbooks, you also have access to standard Ansible facts discovered by `setup` module.

Included custom Ansible modules

The provided `playbooks_path/library` folder includes several custom Ansible modules used by default implementation of `deploy` and `prepare` roles. You can use these modules in your playbooks as well.

`stream_url` Streaming download from HTTP(S) source to the disk device directly, tries to be compatible with Ansibles `get_url` module in terms of module arguments. Due to the low level of such operation it is not idempotent.

`ironic_parted` creates partition tables and partitions with `parted` utility. Due to the low level of such operation it is not idempotent. Please read the documentation included in the modules source for more information about this module and its arguments. The name is chosen so that the `parted` module included in Ansible is not shadowed.

Anaconda deploy

The `anaconda` deploy interface is another option for highly customized deployments. See *Deploying with anaconda deploy interface* for more details.

Ramdisk deploy

The ramdisk interface is intended to provide a mechanism to deploy an instance where the item to be deployed is in reality a ramdisk. It is documented separately, see *Booting a Ramdisk or an ISO*.

Custom agent deploy

The `custom-agent` deploy interface is designed for operators who want to completely orchestrate writing the instance image using in-band deploy steps from a custom agent image. If you use this deploy interface, you are responsible to provide all necessary deploy steps with priorities between 61 and 99 (see *Agent steps* for information on priorities).

4.1.2 Hardware Types

iBMC driver

Overview

The `ibmc` driver is targeted for Huawei V5 series rack server such as 2288H V5, CH121 V5. The iBMC hardware type enables the user to take advantage of features of Huawei iBMC to control Huawei server.

The `ibmc` hardware type supports the following Ironic interfaces:

- Management Interface: Boot device management
- Power Interface: Power management
- *RAID Interface*: RAID controller and disk management
- *Vendor Interface*: ibmc passthru interfaces

Prerequisites

The HUAWEI iBMC Client library should be installed on the ironic conductor node(s).

For example, it can be installed with `pip`:

```
sudo pip install python-ibmcclient
```

Enabling the iBMC driver

1. Add `ibmc` to the list of `enabled_hardware_types`, `enabled_power_interfaces`, `enabled_vendor_interfaces` and `enabled_management_interfaces` in `/etc/ironic/ironic.conf`. For example:

```
[DEFAULT]
...
enabled_hardware_types = ibmc
enabled_power_interfaces = ibmc
enabled_management_interfaces = ibmc
enabled_raid_interfaces = ibmc
enabled_vendor_interfaces = ibmc
```

2. Restart the ironic conductor service:

```
sudo service ironic-conductor restart

# Or, for RDO:
sudo systemctl restart openstack-ironic-conductor
```

Registering a node with the iBMC driver

Nodes configured to use the driver should have the `driver` property set to `ibmc`.

The following properties are specified in the nodes `driver_info` field:

- `ibmc_address`:

  The URL address to the ibmc controller. It must include the authority portion of the URL, and can optionally include the scheme. If the scheme is missing, https is assumed. For example: https://ibmc.example.com. This is required.

- `ibmc_username`:

  User account with admin/server-profile access privilege. This is required.

- `ibmc_password`:

  User account password. This is required.

- `ibmc_verify_ca`:

  If ibmc_address has the **https** scheme, the driver will use a secure (TLS) connection when talking to the ibmc controller. By default (if this is set to True), the driver will try to verify the host certificates. This can be set to the path of a certificate file or directory with trusted certificates that the driver will use for verification. To disable verifying TLS, set this to False. This is optional.

The `baremetal node create` command can be used to enroll a node with the `ibmc` driver. For example:

```
baremetal node create --driver ibmc
  --driver-info ibmc_address=https://example.com \
  --driver-info ibmc_username=admin \
  --driver-info ibmc_password=password
```

For more information about enrolling nodes see *Enrollment* in the install guide.

RAID Interface

Currently, only RAID controller which supports OOB management can be managed.

See *RAID Configuration* for more information on Ironic RAID support.

The following properties are supported by the iBMC raid interface implementation, `ibmc`:

Mandatory properties

- `size_gb`: Size in gigabytes (integer) for the logical disk. Use `MAX` as `size_gb` if this logical disk is supposed to use the rest of the space available.
- `raid_level`: RAID level for the logical disk. Valid values are `JBOD`, `0`, `1`, `5`, `6`, `1+0`, `5+0` and `6+0`. And it is possible that some RAID controllers can only support a subset RAID levels.

Note: RAID level `2` is not supported by `iBMC` driver.

Optional properties

- `is_root_volume`: Optional. Specifies whether this disk is a root volume. By default, this is `False`.
- `volume_name`: Optional. Name of the volume to be created. If this is not specified, it will be N/A.

Backing physical disk hints

See *RAID Configuration* for more information on backing disk hints.

These are machine-independent properties. The hints are specified for each logical disk to help Ironic find the desired disks for RAID configuration.

- `share_physical_disks`
- `disk_type`
- `interface_type`
- `number_of_physical_disks`

Backing physical disks

These are HUAWEI RAID controller dependent properties:

- `controller`: Optional. Supported values are: RAID storage id, RAID storage name or RAID controller name. If a bare metal server have more than one controller, this is mandatory. Typical values would look like:
  - RAID Storage Id: `RAIDStorage0`
  - RAID Storage Name: `RAIDStorage0`
  - RAID Controller Name: `RAID Card1 Controller`.
- `physical_disks`: Optional. Supported values are: disk-id, disk-name or disk serial number. Typical values for hdd disk would look like:
  - Disk Id: `HDDPlaneDisk0`
  - Disk Name: `Disk0`.
  - Disk SerialNumber: `38DGK77LF77D`

Delete RAID configuration

For `delete_configuration` step, `ibmc` will do:

- delete all logical disks
- delete all hot-spare disks

Logical disks creation priority

Logical Disks creation priority based on three properties:

- `share_physical_disks`
- `physical_disks`
- `size_gb`

The logical disks creation priority strictly follow the table below, if multiple logical disks have the same priority, then they will be created with the same order in `logical_disks` array.

Share physical disks	Specified Physical Disks	Size
no	yes	int|max
no	no	int
yes	yes	int
yes	yes	max
yes	no	int
yes	no	max
no	no	max

Physical disks choice strategy

Note: physical-disk-group: a group of physical disks which have been used by some logical-disks with same RAID level.

- If no `physical_disks` are specified, the waste least strategy will be used to choose the physical disks.
  - waste least disk capacity: when using disks with different capacity, it will cause a waste of disk capacity. This is to avoid with highest priority.
  - using least total disk capacity: for example, we can create 400G RAID 5 with both 5 100G-disks and 3 200G-disks. 5 100G disks is a better strategy because it uses a 500G capacity totally. While 3 200G-disks are 600G totally.
  - using least disk count: finally, if waste capacity and total disk capacity are both the same (it rarely happens?), we will choose the one with the minimum number of disks.
- when `share_physical_disks` option is present, `ibmc` driver will create logical disk upon existing physical-disk-group list first. Only when no existing physical-disk-group matches, then it chooses unused physical disks with same strategy described above. When multiple exists physical-disk-groups matches, it will use waste least strategy too, the bigger capacity left the better. For example, to create a logical disk shown below on a `ibmc` server which has two RAID5 logical disks already. And the shareable capacity of this two logical-disks are 500G and 300G, then `ibmc` driver will choose the second one.

```
{
   "logical_disks": [
       {
           "controller": "RAID Card1 Controller",
           "raid_level": "5",
           "size_gb": 100,
           "share_physical_disks": true
       }
   ]
}
```

  And the `ibmc` server has two RAID5 logical disks already.

- When `size_gb` is set to `MAX`, `ibmc` driver will auto work through all possible cases and choose the best solution which has the biggest capacity and use least capacity. For example: to create a RAID 5+0 logical disk with MAX size in a server has 9 200G-disks, it will finally choose 8 disks + span-number 2 but not 9 disks + span-number 3. Although they both have 1200G capacity totally, but the former uses only 8 disks and the latter uses 9 disks. If you want to choose the latter solution, you can specified the disk count to use by adding `number_of_physical_disks` option.

```
{
   "logical_disks": [
       {
           "controller": "RAID Card1 Controller",
           "raid_level": "5+0",
           "size_gb": "MAX"
```

(continues on next page)

(continued from previous page)

```
        }
    ]
}
```

Examples

In a typical scenario we may want to create:

- RAID 5, 500G, root OS volume with 3 disks
- RAID 5, rest available space, data volume with rest disks

```
{
  "logical_disks": [
      {
          "volume_name": "os_volume",
          "controller": "RAID Card1 Controller",
          "is_root_volume": "True",
          "physical_disks": [
              "Disk0",
              "Disk1",
              "Disk2"
          ],
          "raid_level": "5",
          "size_gb": "500"
      },
      {
          "volume_name": "data_volume",
          "controller": "RAID Card1 Controller",
          "raid_level": "5",
          "size_gb": "MAX"
      }
  ]
}
```

Vendor Interface

The `ibmc` hardware type provides vendor passthru interfaces shown below:

Method Name	HTTP Method	Description
boot_up_seq	GET	Query boot up sequence
get_raid_controller_list	GET	Query RAID controller summary info

iDRAC driver

Overview

The integrated Dell Remote Access Controller (iDRAC) is an out-of-band management platform on Dell EMC servers, and is supported directly by the `idrac` hardware type. This driver uses the Dell Web Services for Management (WSMAN) protocol and the standard Distributed Management Task Force (DMTF) Redfish protocol to perform all of its functions.

iDRAC hardware is also supported by the generic `ipmi` and `redfish` hardware types, though with smaller feature sets.

Key features of the Dell iDRAC driver include:

- Out-of-band node inspection
- Boot device management and firmware management
- Power management
- RAID controller management and RAID volume configuration
- BIOS settings configuration

Ironic Features

The `idrac` hardware type supports the following Ironic interfaces:

- *BIOS Interface*: BIOS management
- *Inspect Interface*: Hardware inspection
- *Management Interface*: Boot device and firmware management
- Power Interface: Power management
- *RAID Interface*: RAID controller and disk management
- *Vendor Interface*: BIOS management (WSMAN) and eject virtual media (Redfish)

Prerequisites

The `idrac` hardware type requires the `python-dracclient` library to be installed on the ironic conductor node(s) if an Ironic node is configured to use an `idrac-wsman` interface implementation, for example:

```
sudo pip install 'python-dracclient>=3.1.0'
```

Additionally, the `idrac` hardware type requires the `sushy` library to be installed on the ironic conductor node(s) if an Ironic node is configured to use an `idrac-redfish` interface implementation, for example:

```
sudo pip install 'python-dracclient>=3.1.0' 'sushy>=2.0.0'
```

Enabling

The iDRAC driver supports WSMAN for the bios, inspect, management, power, raid, and vendor interfaces. In addition, it supports Redfish for the bios, inspect, management, power, and raid interfaces. The iDRAC driver allows you to mix and match WSMAN and Redfish interfaces.

The `idrac-wsman` implementation must be enabled to use WSMAN for an interface. The `idrac-redfish` implementation must be enabled to use Redfish for an interface.

To enable the `idrac` hardware type with the minimum interfaces, all using WSMAN, add the following to your `/etc/ironic/ironic.conf`:

```
[DEFAULT]
enabled_hardware_types=idrac
enabled_management_interfaces=idrac-wsman
enabled_power_interfaces=idrac-wsman
```

To enable all optional features (BIOS, inspection, RAID, and vendor passthru) using Redfish where it is supported and WSMAN where not, use the following configuration:

```
[DEFAULT]
enabled_hardware_types=idrac
enabled_bios_interfaces=idrac-redfish
enabled_inspect_interfaces=idrac-redfish
enabled_management_interfaces=idrac-redfish
enabled_power_interfaces=idrac-redfish
enabled_raid_interfaces=idrac-redfish
enabled_vendor_interfaces=idrac-redfish
```

Below is the list of supported interface implementations in priority order:

Interface	Supported Implementations
`bios`	`idrac-wsman`, `idrac-redfish`, `no-bios`
`boot`	`ipxe`, `pxe`, `idrac-redfish-virtual-media`
`console`	`no-console`
`deploy`	`direct`, `ansible`, `ramdisk`
`inspect`	`idrac-wsman`, `idrac`, `idrac-redfish`, `inspector`, `no-inspect`
`management`	`idrac-wsman`, `idrac`, `idrac-redfish`
`network`	`flat`, `neutron`, `noop`
`power`	`idrac-wsman`, `idrac`, `idrac-redfish`
`raid`	`idrac-wsman`, `idrac`, `idrac-redfish`, `no-raid`
`rescue`	`no-rescue`, `agent`
`storage`	`noop`, `cinder`, `external`
`vendor`	`idrac-wsman`, `idrac`, `idrac-redfish`, `no-vendor`

Note: `idrac` is the legacy name of the WSMAN interface. It has been deprecated in favor of `idrac-wsman` and may be removed in a future release.

Protocol-specific Properties

The WSMAN and Redfish protocols require different properties to be specified in the Ironic nodes `driver_info` field to communicate with the bare metal systems iDRAC.

The WSMAN protocol requires the following properties:

- `drac_username`: The WSMAN user name to use when communicating with the iDRAC. Usually `root`.
- `drac_password`: The password for the WSMAN user to use when communicating with the iDRAC.
- `drac_address`: The IP address of the iDRAC.

The Redfish protocol requires the following properties:

- `redfish_username`: The Redfish user name to use when communicating with the iDRAC. Usually `root`.
- `redfish_password`: The password for the Redfish user to use when communicating with the iDRAC.
- `redfish_address`: The URL address of the iDRAC. It must include the authority portion of the URL, and can optionally include the scheme. If the scheme is missing, https is assumed.
- `redfish_system_id`: The Redfish ID of the server to be managed. This should always be: `/redfish/v1/Systems/System.Embedded.1`.

For other Redfish protocol parameters see *Redfish driver*.

If using only interfaces which use WSMAN (`idrac-wsman`), then only the WSMAN properties must be supplied. If using only interfaces which use Redfish (`idrac-redfish`), then only the Redfish properties must be supplied. If using a mix of interfaces, where some use WSMAN and others use Redfish, both the WSMAN and Redfish properties must be supplied.

Enrolling

The following command enrolls a bare metal node with the `idrac` hardware type using WSMAN for all interfaces:

```
baremetal node create --driver idrac \
    --driver-info drac_username=user \
    --driver-info drac_password=pa$$w0rd \
    --driver-info drac_address=drac.host
```

The following command enrolls a bare metal node with the `idrac` hardware type using Redfish for all interfaces:

```
baremetal node create --driver idrac \
    --driver-info redfish_username=user \
    --driver-info redfish_password=pa$$w0rd \
    --driver-info redfish_address=drac.host \
    --driver-info redfish_system_id=/redfish/v1/Systems/System.Embedded.1 \
    --bios-interface idrac-redfish \
```

(continues on next page)

(continued from previous page)

```
    --inspect-interface idrac-redfish \
    --management-interface idrac-redfish \
    --power-interface idrac-redfish \
    --raid-interface idrac-redfish \
    --vendor-interface idrac-redfish
```

The following command enrolls a bare metal node with the `idrac` hardware type assuming a mix of Redfish and WSMAN interfaces are used:

```
baremetal node create --driver idrac \
    --driver-info drac_username=user \
    --driver-info drac_password=pa$$w0rd
    --driver-info drac_address=drac.host \
    --driver-info redfish_username=user \
    --driver-info redfish_password=pa$$w0rd \
    --driver-info redfish_address=drac.host \
    --driver-info redfish_system_id=/redfish/v1/Systems/System.Embedded.1 \
    --bios-interface idrac-redfish \
    --inspect-interface idrac-redfish \
    --management-interface idrac-redfish \
    --power-interface idrac-redfish
```

Note: If using WSMAN for the management interface, then WSMAN must be used for the power interface. The same applies to Redfish. It is currently not possible to use Redfish for one and WSMAN for the other.

BIOS Interface

The BIOS interface implementations supported by the `idrac` hardware type allows BIOS to be configured with the standard clean/deploy step approach.

Example

A clean step to enable `Virtualization` and `SRIOV` in BIOS of an iDRAC BMC would be as follows:

```
{
  "target":"clean",
  "clean_steps": [
    {
      "interface": "bios",
      "step": "apply_configuration",
      "args": {
        "settings": [
          {
            "name": "ProcVirtualization",
            "value": "Enabled"
```

(continues on next page)

(continued from previous page)

```
            },
            {
              "name": "SriovGlobalEnable",
              "value": "Enabled"
            }
          ]
        }
      }
    ]
}
```

See the *Known Issues* for a known issue with `factory_reset` clean step. For additional details of BIOS configuration, see *BIOS Configuration*.

Inspect Interface

The Dell iDRAC out-of-band inspection process catalogs all the same attributes of the server as the IPMI driver. Unlike IPMI, it does this without requiring the system to be rebooted, or even to be powered on. Inspection is performed using the Dell WSMAN or Redfish protocol directly without affecting the operation of the system being inspected.

The inspection discovers the following properties:

- `cpu_arch`: cpu architecture
- `cpus`: number of cpus
- `local_gb`: disk size in gigabytes
- `memory_mb`: memory size in megabytes

Extra capabilities:

- `boot_mode`: UEFI or BIOS boot mode.
- `pci_gpu_devices`: number of GPU devices connected to the bare metal.

It also creates baremetal ports for each NIC port detected in the system. The `idrac-wsman` inspect interface discovers which NIC ports are configured to PXE boot and sets `pxe_enabled` to `True` on those ports. The `idrac-redfish` inspect interface does not currently set `pxe_enabled` on the ports. The user should ensure that `pxe_enabled` is set correctly on the ports following inspection with the `idrac-redfish` inspect interface.

Management Interface

The management interface for `idrac-redfish` supports:

- updating firmware on nodes using a manual cleaning step. See *Redfish driver* for more information on firmware update support.
- updating system and getting its inventory using configuration molds. For more information see *Import and export configuration*.

Import and export configuration

The clean and deploy steps provided in this section allow to configure the system and collect the system inventory using configuration mold files.

The introduction of this feature in the Wallaby release is experimental.

These steps are:

- `export_configuration` with the `export_configuration_location` input parameter to export the configuration from the existing system.
- `import_configuration` with the `import_configuration_location` input parameter to import the existing configuration mold into the system.
- `import_export_configuration` with the `export_configuration_location` and `import_configuration_location` input parameters. This step combines the previous two steps into one step that first imports existing configuration mold into system, then exports the resulting configuration.

The input parameters provided include the URL where the configuration mold is to be stored after the export, or the reference location for an import. For more information on setting up storage and available options see *Storage setup*.

Configuration molds are JSON files that contain three top-level sections: `bios`, `raid` and `oem`. The following is an example of a configuration mold:

```
{
  "bios": {
    "reset": false,
    "settings": [
      {
        "name": "ProcVirtualization",
        "value": "Enabled"
      },
      {
        "name": "MemTest",
        "value": "Disabled"
      }
    ]
  }
  "raid": {
    "create_nonroot_volumes": true,
    "create_root_volume": true,
    "delete_existing": false,
    "target_raid_config": {
      "logical_disks": [
        {
          "size_gb": 50,
          "raid_level": "1+0",
          "controller": "RAID.Integrated.1-1",
          "volume_name": "root_volume",
          "is_root_volume": true,
          "physical_disks": [
```

(continues on next page)

(continued from previous page)

```
            "Disk.Bay.0:Encl.Int.0-1:RAID.Integrated.1-1",
            "Disk.Bay.1:Encl.Int.0-1:RAID.Integrated.1-1"
          ]
        },
        {
          "size_gb": 100,
          "raid_level": "5",
          "controller": "RAID.Integrated.1-1",
          "volume_name": "data_volume",
          "physical_disks": [
            "Disk.Bay.2:Encl.Int.0-1:RAID.Integrated.1-1",
            "Disk.Bay.3:Encl.Int.0-1:RAID.Integrated.1-1",
            "Disk.Bay.4:Encl.Int.0-1:RAID.Integrated.1-1"
          ]
        }
      ]
    }
  }
  "oem": {
    "interface": "idrac-redfish",
    "data": {
      "SystemConfiguration": {
        "Model": "PowerEdge R640",
        "ServiceTag": "8CY9Z99",
        "TimeStamp": "Fri Jun 26 08:43:15 2020",
        "Components": [
          {
            [...]
            "FQDD": "NIC.Slot.1-1-1",
            "Attributes": [
              {
              "Name": "BlnkLeds",
              "Value": "15",
              "Set On Import": "True",
              "Comment": "Read and Write"
              },
              {
              "Name": "VirtMacAddr",
              "Value": "00:00:00:00:00:00",
              "Set On Import": "False",
              "Comment": "Read and Write"
              },
              {
              "Name": "VirtualizationMode",
              "Value": "NONE",
              "Set On Import": "True",
              "Comment": "Read and Write"
              },
            [...]
```

(continues on next page)

(continued from previous page)

```
            ]
        }
    ]
}
}
}
```

Currently, the OEM section is the only section that is supported. The OEM section uses the iDRAC Server Configuration Profile (SCP) and can be edited as necessary if it complies with the SCP. For more information about SCP and its capabilities, see SCP_Reference_Guide.

Note: iDRAC BMC connection settings are not exported to avoid overwriting these in another system when using unmodified exported configuration mold in import step. If need to replicate iDRAC BMC connection settings, then add these settings manually to configuration mold for import step.

To replicate the system configuration to that of a similar system, perform the following steps:

1. Configure a golden, or one to many, system.
2. Use the `export_configuration` step to export the configuration to the wanted location.
3. Adjust the exported configuration mold for other systems to replicate. For example, remove sections that do not need to be replicated such as iDRAC connection settings. The configuration mold can be accessed directly from the storage location.
4. Import the selected configuration mold into the other systems using the `import_configuration` step.

It is not mandatory to use `export_configuration` step to create a configuration mold. Upload the file to a designated storage location without using Ironic if it has been created manually or by other means.

Storage setup

To start using these steps, configure the storage location. The settings can be found in the `[molds]` section. Configure the storage type from the `[molds]storage` setting. Currently, `swift`, which is enabled by default, and `http` are supported.

In the setup input parameters, the complete HTTP URL is used. This requires that the containers (for `swift`) and the directories (for `http`) are created beforehand, and that read/write access is configured accordingly.

Note: Use of TLS is strongly advised.

This setup configuration allows a user to access these locations outside of Ironic to list, create, update, and delete the configuration molds.

For more information see *Swift configuration* and *HTTP configuration*.

Swift configuration

To use Swift with configuration molds,

1. Create the containers to be used for configuration mold storage.
2. For Ironic Swift user that is configured in the `[swift]` section add read/write access to these containers.

HTTP configuration

To use HTTP server with configuration molds,

1. Enable HTTP PUT support.
2. Create the directory to be used for the configuration mold storage.
3. Configure read/write access for HTTP Basic access authentication and provide user credentials in `[molds]user` and `[molds]password` fields.

The HTTP web server does not support multitenancy and is intended to be used in a stand-alone Ironic, or single-tenant OpenStack environment.

RAID Interface

See *RAID Configuration* for more information on Ironic RAID support.

RAID interface of `redfish` hardware type can be used on iDRAC systems. Compared to `redfish` RAID interface, using `idrac-redfish` adds:

- Waiting for real-time operations to be available on RAID controllers. When using `redfish` this is not guaranteed and reboots might be intermittently required to complete,
- Converting non-RAID disks to RAID mode if there are any,
- Clearing foreign configuration, if any, after deleting virtual disks.

The following properties are supported by the iDRAC WSMAN and Redfish RAID interface implementation:

Note: When using `idrac-redfish` for RAID interface iDRAC firmware greater than 4.40.00.00 is required.

Mandatory properties

- `size_gb`: Size in gigabytes (integer) for the logical disk. Use `MAX` as `size_gb` if this logical disk is supposed to use the rest of the space available.
- `raid_level`: RAID level for the logical disk. Valid values are `0`, `1`, `5`, `6`, `1+0`, `5+0` and `6+0`.

Note: `JBOD` and `2` are not supported, and will fail with reason: Cannot calculate spans for RAID level.

Optional properties

- `is_root_volume`: Optional. Specifies whether this disk is a root volume. By default, this is `False`.
- `volume_name`: Optional. Name of the volume to be created. If this is not specified, it will be auto-generated.

Backing physical disk hints

See *RAID Configuration* for more information on backing disk hints.

These are machine-independent information. The hints are specified for each logical disk to help Ironic find the desired disks for RAID configuration.

- `disk_type`
- `interface_type`
- `share_physical_disks`
- `number_of_physical_disks`

Backing physical disks

These are Dell RAID controller-specific values and must match the names provided by the iDRAC.

- `controller`: Mandatory. The name of the controller to use.
- `physical_disks`: Optional. The names of the physical disks to use.

Note: `physical_disks` is a mandatory parameter if the property `size_gb` is set to `MAX`.

Examples

Creation of RAID `1+0` logical disk with six disks on one controller:

```
{ "logical_disks":
  [ { "controller": "RAID.Integrated.1-1",
      "is_root_volume": "True",
      "physical_disks": [
        "Disk.Bay.0:Enclosure.Internal.0-1:RAID.Integrated.1-1",
        "Disk.Bay.1:Enclosure.Internal.0-1:RAID.Integrated.1-1",
        "Disk.Bay.2:Enclosure.Internal.0-1:RAID.Integrated.1-1",
        "Disk.Bay.3:Enclosure.Internal.0-1:RAID.Integrated.1-1",
        "Disk.Bay.4:Enclosure.Internal.0-1:RAID.Integrated.1-1",
        "Disk.Bay.5:Enclosure.Internal.0-1:RAID.Integrated.1-1"],
      "raid_level": "1+0",
      "size_gb": "MAX"}]}
```

Manual RAID Invocation

The following command can be used to delete any existing RAID configuration. It deletes all virtual disks/RAID volumes, unassigns all global and dedicated hot spare physical disks, and clears foreign configuration:

```
baremetal node clean --clean-steps \
  '[{"interface": "raid", "step": "delete_configuration"}]' ${node_uuid}
```

The following command shows an example of how to set the target RAID configuration:

```
baremetal node set --target-raid-config '{ "logical_disks":
  [ { "controller": "RAID.Integrated.1-1",
      "is_root_volume": true,
      "physical_disks": [
        "Disk.Bay.0:Enclosure.Internal.0-1:RAID.Integrated.1-1",
        "Disk.Bay.1:Enclosure.Internal.0-1:RAID.Integrated.1-1"],
      "raid_level": "0",
      "size_gb": "MAX"}]}' ${node_uuid}
```

The following command can be used to create a RAID configuration:

```
baremetal node clean --clean-steps \
  '[{"interface": "raid", "step": "create_configuration"}]' <node>
```

When the physical disk names or controller names are not known, the following Python code example shows how the `python-dracclient` can be used to fetch the information directly from the Dell bare metal:

```
import dracclient.client


client = dracclient.client.DRACClient(
    host="192.168.1.1",
    username="root",
    password="calvin")
controllers = client.list_raid_controllers()
print(controllers)

physical_disks = client.list_physical_disks()
print(physical_disks)
```

Or using `sushy` with Redfish:

```
import sushy


client = sushy.Sushy('https://192.168.1.1', username='root', password='calvin
↪', verify=False)
for s in client.get_system_collection().get_members():
  print("System: %(id)s" % {'id': s.identity})
```

(continues on next page)

(continued from previous page)

```
  for c in system1.storage.get_members():
      print("\tController: %(id)s" % {'id': c.identity})
      for d in c.drives:
        print("\t\tDrive: %(id)s" % {'id': d.identity})
```

Vendor Interface

idrac-wsman

Dell iDRAC BIOS management is available through the Ironic WSMAN vendor passthru interface.

Method Name	HTTP Method	Description
abandon_bios_config	DELETE	Abandon a BIOS configuration job.
commit_bios_config	POST	Commit a BIOS configuration job submitted through `set_bios_config`. Required argument: `reboot` - indicates whether a reboot job should be automatically created with the config job. Returns a dictionary containing the `job_id` key with the ID of the newly created config job, and the `reboot_required` key indicating whether the node needs to be rebooted to execute the config job.
get_bios_config	GET	Returns a dictionary containing the nodes BIOS settings.
list_unfinished_jobs	GET	Returns a dictionary containing the key `unfinished_jobs`; its value is a list of dictionaries. Each dictionary represents an unfinished config job object.
set_bios_config	POST	Change the BIOS configuration on a node. Required argument: a dictionary of `{AttributeName: NewValue}`. Returns a dictionary containing the `is_commit_required` key indicating whether `commit_bios_config` needs to be called to apply the changes and the `is_reboot_required` value indicating whether the server must also be rebooted. Possible values are `true` and `false`.

Examples

Get BIOS Config

```
baremetal node passthru call --http-method GET <node> get_bios_config
```

Snippet of output showing virtualization enabled:

```
{"ProcVirtualization": {
      "current_value": "Enabled",
      "instance_id": "BIOS.Setup.1-1:ProcVirtualization",
      "name": "ProcVirtualization",
      "pending_value": null,
      "possible_values": [
          "Enabled",
          "Disabled"],
      "read_only": false }}
```

There are a number of items to note from the above snippet:

- `name`: this is the name to use in a call to `set_bios_config`.
- `current_value`: the current state of the setting.
- `pending_value`: if the value has been set, but not yet committed, the new value is shown here. The change can either be committed or abandoned.
- `possible_values`: shows a list of valid values which can be used in a call to `set_bios_config`.
- `read_only`: indicates if the value is capable of being changed.

Set BIOS Config

```
baremetal node passthru call <node> set_bios_config --arg "name=value"
```

Walkthrough of perfoming a BIOS configuration change:

The following section demonstrates how to change BIOS configuration settings, detect that a commit and reboot are required, and act on them accordingly. The two properties that are being changed are:

- Enable virtualization technology of the processor
- Globally enable SR-IOV

```
baremetal node passthru call <node> set_bios_config \
  --arg "ProcVirtualization=Enabled" \
  --arg "SriovGlobalEnable=Enabled"
```

This returns a dictionary indicating what actions are required next:

```
{
  "is_reboot_required": true,
  "is_commit_required": true
}
```

Commit BIOS Changes

The next step is to commit the pending change to the BIOS. Note that in this example, the `reboot` argument is set to `true`. The response indicates that a reboot is no longer required as it has been scheduled automatically by the `commit_bios_config` call. If the reboot argument is not supplied, the job is still created, however it remains in the `scheduled` state until a reboot is performed. The reboot can be initiated through the Ironic power API.

```
baremetal node passthru call <node> commit_bios_config \
  --arg "reboot=true"
```

```
{
  "job_id": "JID_499377293428",
  "reboot_required": false
}
```

The state of any executing job can be queried:

```
baremetal node passthru call --http-method GET <node> list_unfinished_jobs
```


```
{"unfinished_jobs":
    [{"status": "Scheduled",
      "name": "ConfigBIOS:BIOS.Setup.1-1",
      "until_time": "TIME_NA",
      "start_time": "TIME_NOW",
      "message": "Task successfully scheduled.",
      "percent_complete": "0",
      "id": "JID_499377293428"}]}
```


Abandon BIOS Changes

Instead of committing, a pending change can be abandoned:

```
baremetal node passthru call --http-method DELETE <node> abandon_bios_config
```

The abandon command does not provide a response body.

Change Boot Mode

The boot mode of the iDRAC can be changed to:

- BIOS - Also called legacy or traditional boot mode. The BIOS initializes the systems processors, memory, bus controllers, and I/O devices. After initialization is complete, the BIOS passes control to operating system (OS) software. The OS loader uses basic services provided by the system BIOS to locate and load OS modules into system memory. After booting the system, the BIOS and embedded management controllers execute system management algorithms, which monitor and optimize the condition of the underlying hardware. BIOS configuration settings enable fine-tuning of the performance, power management, and reliability features of the system.
- UEFI - The Unified Extensible Firmware Interface does not change the traditional purposes of the system BIOS. To a large extent, a UEFI-compliant BIOS performs the same initialization, boot, configuration, and management tasks as a traditional BIOS. However, UEFI does change the interfaces and data structures the BIOS uses to interact with I/O device firmware and operating system software. The primary intent of UEFI is to eliminate shortcomings in the traditional BIOS environment, enabling system firmware to continue scaling with industry trends.

The UEFI boot mode offers:

- Improved partitioning scheme for boot media
- Support for media larger than 2 TB
- Redundant partition tables
- Flexible handoff from BIOS to OS
- Consolidated firmware user interface
- Enhanced resource allocation for boot device firmware

The boot mode can be changed via the WSMAN vendor passthru interface as follows:

```
baremetal node passthru call <node> set_bios_config \
  --arg "BootMode=Uefi"

baremetal node passthru call <node> commit_bios_config \
  --arg "reboot=true"
```

```
baremetal node passthru call <node> set_bios_config \
  --arg "BootMode=Bios"

baremetal node passthru call <node> commit_bios_config \
  --arg "reboot=true"
```

idrac-redfish

Through the `idrac-redfish` vendor passthru interface these methods are available:

Method Name	HTTP Method	Description
`eject_media`	POST	Eject a virtual media device. If no device is provided then all attached devices will be ejected. Optional argument: `boot_device` - the boot device to eject, either, `cd`, `dvd`, `usb` or `floppy`.

Known Issues

Nodes go into maintenance mode

After some period of time, nodes managed by the `idrac` hardware type may go into maintenance mode in Ironic. This issue can be worked around by changing the Ironic power state poll interval to 70 seconds. See `[conductor]sync_power_state_interval` in `/etc/ironic/ironic.conf`.

PXE reset with factory_reset BIOS clean step

When using the `UEFI boot mode` with non-default PXE interface, the factory reset can cause the PXE interface to be reset to default, which doesnt allow the server to PXE boot for any further operations. This can cause a `clean_failed` state on the node or `deploy_failed` if you attempt to deploy a node after this step. For now, the only solution is for the operator to manually restore the PXE settings of the server for it to PXE boot again, properly. The problem is caused by the fact that with the `UEFI boot mode`, the `idrac` uses BIOS settings to manage PXE configuration. This is not the case with the `BIOS boot mode` where the PXE configuration is handled as a configuration job on the integrated NIC itself, independently of the BIOS settings.

WSMAN vendor passthru timeout

When iDRAC is not ready and executing WSMAN vendor passthru commands, they take more time as waiting for iDRAC to become ready again and then time out, for example:

```
baremetal node passthru call --http-method GET \
  aed58dca-1b25-409a-a32f-3a817d59e1e0 list_unfinished_jobs
Timed out waiting for a reply to message ID 547ce7995342418c99ef1ea4a0054572␣
↪(HTTP 500)
```

To avoid this need to increase timeout for messaging in `/etc/ironic/ironic.conf` and restart Ironic API service.

```
[DEFAULT]
rpc_response_timeout = 600
```

Timeout when powering off

Some servers might be slow when soft powering off and time out. The default retry count is 6, resulting in 30 seconds timeout (the default retry interval set by `post_deploy_get_power_state_retry_interval` is 5 seconds). To resolve this issue, increase the timeout to 90 seconds by setting the retry count to 18 as follows:

```
[agent]
post_deploy_get_power_state_retries = 18
```

Unable to mount remote share with iDRAC firmware before 4.40.40.00

When using iDRAC firmware 4.40.00.00 and consecutive versions before 4.40.40.00 with virtual media boot and new Virtual Console plug-in type eHTML5, there is an error: Unable to mount remote share. This is a known issue that is fixed in 4.40.40.00 iDRAC firmware release. If cannot upgrade, then adjust settings in iDRAC to use plug-in type HTML5. In iDRAC web UI go to Configuration -> Virtual Console and select Plug-in Type to HTML5.

During upgrade to 4.40.00.00 or newer iDRAC firmware eHTML5 is automatically selected if default plug-in type has been used and never changed. Systems that have plug-in type changed will keep selected plug-in type after iDRAC firmware upgrade.

Firmware update from Swift fails before 6.00.00.00

With iDRAC firmware prior to 6.00.00.00 and when using Swift to stage firmware update files in Management interface `firmware_update` clean step of `redfish` or `idrac` hardware type, the cleaning fails with error An internal error occurred. Unable to complete the specified operation. in iDRAC job. This is fixed in iDRAC firmware 6.00.00.00. If cannot upgrade, then use HTTP service to stage firmware files for iDRAC.

iLO driver

Overview

iLO driver enables to take advantage of features of iLO management engine in HPE ProLiant servers. The `ilo` hardware type is targeted for HPE ProLiant Gen8 and Gen9 systems which have iLO 4 management engine. From **Pike** release `ilo` hardware type supports ProLiant Gen10 systems which have iLO 5 management engine. iLO5 conforms to Redfish API and hence hardware type `redfish` (see *Redfish driver*) is also an option for this kind of hardware but it lacks the iLO specific features.

For more details and for up-to-date information (like tested platforms, known issues, etc), please check the iLO driver wiki page.

For enabling Gen10 systems and getting detailed information on Gen10 feature support in Ironic please check this Gen10 wiki section.

Hardware type

ProLiant hardware is primarily supported by the `ilo` hardware type. `ilo5` hardware type is only supported on ProLiant Gen10 and later systems. Both hardware can be used with reference hardware type `ipmi` (see *IPMI driver*) and `redfish` (see *Redfish driver*). For information on how to enable the `ilo` and `ilo5` hardware type, see *Enabling hardware types*.

Note: Only HPE ProLiant Gen10 servers supports hardware type `redfish`.

Warning: It is important to note that while the HPE Edgeline series of servers may contain iLO adapters, they are known to not be compatible with the `ilo` hardware type. The `redfish` hardware type should be used instead.

The hardware type `ilo` supports following HPE server features:

- *Boot mode support*
- *UEFI Secure Boot Support*
- *Node Cleaning Support*
- *Node Deployment Customization*
- *Hardware Inspection Support*
- *Swiftless deploy for intermediate images*
- *HTTP(S) Based Deploy Support*
- *Support for iLO driver with Standalone Ironic*
- *RAID Support*
- *Disk Erase Support*
- *Initiating firmware update as manual clean step*
- *Smart Update Manager (SUM) based firmware update*

- *Updating security parameters as manual clean step*
- *Update Minimum Password Length security parameter as manual clean step*
- *Update Authentication Failure Logging security parameter as manual clean step*
- *Activating iLO Advanced license as manual clean step*
- *Removing CA certificates from iLO as manual clean step*
- *Firmware based UEFI iSCSI boot from volume support*
- *Certificate based validation in iLO*
- *Rescue mode support*
- *Inject NMI support*
- *Soft power operation support*
- *BIOS configuration support*
- *IPv6 support*
- *Layer 3 or DHCP-less ramdisk booting*

Apart from above features hardware type `ilo5` also supports following features:

- *Out of Band RAID Support*
- *Out of Band Sanitize Disk Erase Support*
- *Out of Band One Button Secure Erase Support*
- *UEFI-HTTPS Boot support*

Hardware interfaces

The `ilo` hardware type supports following hardware interfaces:

- **bios** Supports `ilo` and `no-bios`. The default is `ilo`. They can be enabled by using the `[DEFAULT]enabled_bios_interfaces` option in `ironic.conf` as given below:

```
[DEFAULT]
enabled_hardware_types = ilo
enabled_bios_interfaces = ilo,no-bios
```

- **boot** Supports `ilo-virtual-media`, `ilo-pxe` and `ilo-ipxe`. The default is `ilo-virtual-media`. The `ilo-virtual-media` interface provides security enhanced PXE-less deployment by using iLO virtual media to boot up the bare metal node. The `ilo-pxe` and `ilo-ipxe` interfaces use PXE and iPXE respectively for deployment(just like *PXE boot*). These interfaces do not require iLO Advanced license. They can be enabled by using the `[DEFAULT]enabled_boot_interfaces` option in `ironic.conf` as given below:

```
[DEFAULT]
enabled_hardware_types = ilo
enabled_boot_interfaces = ilo-virtual-media,ilo-pxe,ilo-ipxe
```

- **console** Supports `ilo` and `no-console`. The default is `ilo`. They can be enabled by using the `[DEFAULT]enabled_console_interfaces` option in `ironic.conf` as given below:

```
[DEFAULT]
enabled_hardware_types = ilo
enabled_console_interfaces = ilo,no-console
```

Note: To use `ilo` console interface you need to enable iLO feature IPMI/DCMI over LAN Access on iLO4 and iLO5 management engine.

- **inspect** Supports `ilo` and `inspector`. The default is `ilo`. They can be enabled by using the `[DEFAULT]enabled_inspect_interfaces` option in `ironic.conf` as given below:

```
[DEFAULT]
enabled_hardware_types = ilo
enabled_inspect_interfaces = ilo,inspector
```

Note: Ironic Inspector needs to be configured to use `inspector` as the inspect interface.

- **management** Supports only `ilo`. It can be enabled by using the `[DEFAULT]enabled_management_interfaces` option in `ironic.conf` as given below:

```
[DEFAULT]
enabled_hardware_types = ilo
enabled_management_interfaces = ilo
```

- **power** Supports only `ilo`. It can be enabled by using the `[DEFAULT]enabled_power_interfaces` option in `ironic.conf` as given below:

```
[DEFAULT]
enabled_hardware_types = ilo
enabled_power_interfaces = ilo
```

- **raid** Supports `agent` and `no-raid`. The default is `no-raid`. They can be enabled by using the `[DEFAULT]enabled_raid_interfaces` option in `ironic.conf` as given below:

```
[DEFAULT]
enabled_hardware_types = ilo
enabled_raid_interfaces = agent,no-raid
```

- **storage** Supports `cinder` and `noop`. The default is `noop`. They can be enabled by using the `[DEFAULT]enabled_storage_interfaces` option in `ironic.conf` as given below:

```
[DEFAULT]
enabled_hardware_types = ilo
enabled_storage_interfaces = cinder,noop
```

Note: The storage interface `cinder` is supported only when corresponding boot interface

of the `ilo` hardware type based node is `ilo-pxe` or `ilo-ipxe`. Please refer to *Boot From Volume* for configuring `cinder` as a storage interface.

- **rescue** Supports `agent` and `no-rescue`. The default is `no-rescue`. They can be enabled by using the `[DEFAULT]enabled_rescue_interfaces` option in `ironic.conf` as given below:

```
[DEFAULT]
enabled_hardware_types = ilo
enabled_rescue_interfaces = agent,no-rescue
```

The `ilo5` hardware type supports all the `ilo` interfaces described above, except for `boot` and `raid` interfaces. The details of `boot` and `raid` interfaces is as under:

- **raid** Supports `ilo5` and `no-raid`. The default is `ilo5`. They can be enabled by using the `[DEFAULT]enabled_raid_interfaces` option in `ironic.conf` as given below:

```
[DEFAULT]
enabled_hardware_types = ilo5
enabled_raid_interfaces = ilo5,no-raid
```

- **boot** Supports `ilo-uefi-https` apart from the other boot interfaces supported by `ilo` hardware type. This can be enabled by using the `[DEFAULT]enabled_boot_interfaces` option in `ironic.conf` as given below:

```
[DEFAULT]
enabled_hardware_types = ilo5
enabled_boot_interfaces = ilo-uefi-https,ilo-virtual-media
```

The `ilo` and `ilo5` hardware type support all standard `deploy` and `network` interface implementations, see *Enabling hardware interfaces* for details.

The following command can be used to enroll a ProLiant node with `ilo` hardware type:

```
baremetal node create \
    --driver ilo \
    --deploy-interface direct \
    --raid-interface agent \
    --rescue-interface agent \
    --driver-info ilo_address=<ilo-ip-address> \
    --driver-info ilo_username=<ilo-username> \
    --driver-info ilo_password=<ilo-password> \
    --driver-info deploy_iso=<glance-uuid-of-deploy-iso> \
    --driver-info rescue_iso=<glance-uuid-of-rescue-iso>
```

Note: The fields `deploy_iso` and `rescue_iso` used to be called `ilo_deploy_iso` and `ilo_rescue_iso` before the Xena release.

The following command can be used to enroll a ProLiant node with `ilo5` hardware type:

```
baremetal node create \
    --driver ilo5 \
```

(continues on next page)

(continued from previous page)

```
    --deploy-interface direct \
    --raid-interface ilo5 \
    --rescue-interface agent \
    --driver-info ilo_address=<ilo-ip-address> \
    --driver-info ilo_username=<ilo-username> \
    --driver-info ilo_password=<ilo-password> \
    --driver-info deploy_iso=<glance-uuid-of-deploy-iso> \
    --driver-info rescue_iso=<glance-uuid-of-rescue-iso>
```

Please refer to *Enabling drivers and hardware types* for detailed explanation of hardware type.

Node configuration

- Each node is configured for `ilo` and `ilo5` hardware type by setting the following ironic node objects properties in `driver_info`:
  - `ilo_address`: IP address or hostname of the iLO.
  - `ilo_username`: Username for the iLO with administrator privileges.
  - `ilo_password`: Password for the above iLO user.
  - `client_port`: (optional) Port to be used for iLO operations if you are using a custom port on the iLO. Default port used is 443.
  - `client_timeout`: (optional) Timeout for iLO operations. Default timeout is 60 seconds.
  - `ca_file`: (optional) CA certificate file to validate iLO.
  - `console_port`: (optional) Nodes UDP port for console access. Any unused port on the ironic conductor node may be used. This is required only when `ilo-console` interface is used.
- The following properties are also required in node objects `driver_info` if `ilo-virtual-media` boot interface is used:
  - `deploy_iso`: The glance UUID of the deploy ramdisk ISO image.
  - `instance info/boot_iso` property to be either boot iso Glance UUID or a HTTP(S) URL. This is optional property and is used when `boot_option` is set to `netboot` or `ramdisk`.

    Note: When `boot_option` is set to `ramdisk`, the ironic node must be configured to use `ramdisk` deploy interface. See *Ramdisk deploy* for details.

    Note: The `boot_iso` property used to be called `ilo_boot_iso` before the Xena release.

  - `rescue_iso`: The glance UUID of the rescue ISO image. This is optional property and is used when `rescue` interface is set to `agent`.
- The following properties are also required in node objects `driver_info` if `ilo-pxe` or `ilo-ipxe` boot interface is used:
  - `deploy_kernel`: The glance UUID or a HTTP(S) URL of the deployment kernel.

  - `deploy_ramdisk`: The glance UUID or a HTTP(S) URL of the deployment ramdisk.
  - `rescue_kernel`: The glance UUID or a HTTP(S) URL of the rescue kernel. This is optional property and is used when `rescue` interface is set to `agent`.
  - `rescue_ramdisk`: The glance UUID or a HTTP(S) URL of the rescue ramdisk. This is optional property and is used when `rescue` interface is set to `agent`.

- The following properties are also required in node objects `driver_info` if `ilo-uefi-https` boot interface is used for `ilo5` hardware type:
  - `deploy_kernel`: The glance UUID or a HTTPS URL of the deployment kernel.
  - `deploy_ramdisk`: The glance UUID or a HTTPS URL of the deployment ramdisk.
  - `bootloader`: The glance UUID or a HTTPS URL of the bootloader.
  - `rescue_kernel`: The glance UUID or a HTTPS URL of the rescue kernel. This is optional property and is used when `rescue` interface is set to `agent`.
  - `rescue_ramdisk`: The glance UUID or a HTTP(S) URL of the rescue ramdisk. This is optional property and is used when `rescue` interface is set to `agent`.

Note: `ilo-uefi-https` boot interface is supported by only `ilo5` hardware type. If the images are not hosted in glance, the references must be HTTPS URLs hosted by secure webserver. This boot interface can be used only when the current boot mode is `UEFI`.

Note: The fields `deploy_kernel`, `deploy_ramdisk`, `rescue_kernel` `rescue_ramdisk` and `bootloader` used to have an `ilo_` prefix before the Xena release.

- The following parameters are mandatory in `driver_info` if `ilo-inspect` inspect inteface is used and SNMPv3 inspection (*SNMPv3 Authentication* in HPE iLO4 User Guide) is desired:
  - `snmp_auth_user` : The SNMPv3 user.
  - `snmp_auth_prot_password` : The auth protocol pass phrase.
  - `snmp_auth_priv_password` : The privacy protocol pass phrase.

  The following parameters are optional for SNMPv3 inspection:
  - `snmp_auth_protocol` : The Auth Protocol. The valid values are MD5 and SHA. The iLO default value is MD5.
  - `snmp_auth_priv_protocol` : The Privacy protocol. The valid values are AES and DES. The iLO default value is DES.

Note: If configuration values for `ca_file`, `client_port` and `client_timeout` are not provided in the `driver_info` of the node, the corresponding config variables defined under `[ilo]` section in ironic.conf will be used.

Prerequisites

- proliantutils is a python package which contains a set of modules for managing HPE ProLiant hardware.

  Install `proliantutils` module on the ironic conductor node. Minimum version required is 2.8.0:

```
$ pip install "proliantutils>=2.8.0"
```

- `ipmitool` command must be present on the service node(s) where `ironic-conductor` is running. On most distros, this is provided as part of the `ipmitool` package. Please refer to *Hardware Inspection Support* for more information on recommended version.

Different configuration for ilo hardware type

Glance Configuration

1. Configure Glance image service with its storage backend as Swift.
2. Set a temp-url key for Glance user in Swift. For example, if you have configured Glance with user `glance-swift` and tenant as `service`, then run the below command:

```
swift --os-username=service:glance-swift post -m temp-url-
↪key:mysecretkeyforglance
```

3. Fill the required parameters in the `[glance]` section in `/etc/ironic/ironic.conf`. Normally you would be required to fill in the following details:

```
[glance]
swift_temp_url_key=mysecretkeyforglance
swift_endpoint_url=https://10.10.1.10:8080
swift_api_version=v1
swift_account=AUTH_51ea2fb400c34c9eb005ca945c0dc9e1
swift_container=glance
```

   The details can be retrieved by running the below command:

```
$ swift --os-username=service:glance-swift stat -v | grep -i url

StorageURL:     http://10.10.1.10:8080/v1/AUTH_
↪51ea2fb400c34c9eb005ca945c0dc9e1
Meta Temp-Url-Key: mysecretkeyforglance
```

4. Swift must be accessible with the same admin credentials configured in Ironic. For example, if Ironic is configured with the below credentials in `/etc/ironic/ironic.conf`:

```
[keystone_authtoken]
admin_password = password
admin_user = ironic
admin_tenant_name = service
```

   Ensure `auth_version` in `keystone_authtoken` to 2.

Then, the below command should work.:

```
$ swift --os-username ironic --os-password password --os-tenant-name␣
↪service --auth-version 2 stat

                     Account: AUTH_22af34365a104e4689c46400297f00cb
                  Containers: 2
                     Objects: 18
                       Bytes: 1728346241
Objects in policy "policy-0": 18
  Bytes in policy "policy-0": 1728346241
           Meta Temp-Url-Key: mysecretkeyforglance
                 X-Timestamp: 1409763763.84427
                  X-Trans-Id: tx51de96a28f27401eb2833-005433924b
                Content-Type: text/plain; charset=utf-8
               Accept-Ranges: bytes
```

5. Restart the Ironic conductor service:

```
$ service ironic-conductor restart
```

Web server configuration on conductor

- The HTTP(S) web server can be configured in many ways. For apache web server on Ubuntu, refer here
- Following config variables need to be set in `/etc/ironic/ironic.conf`:
  - `use_web_server_for_images` in `[ilo]` section:

    ```
    [ilo]
    use_web_server_for_images = True
    ```

  - `http_url` and `http_root` in `[deploy]` section:

    ```
    [deploy]
    # Ironic compute node's http root path. (string value)
    http_root=/httpboot

    # Ironic compute node's HTTP server URL. Example:
    # http://192.1.2.3:8080 (string value)
    http_url=http://192.168.0.2:8080
    ```

`use_web_server_for_images`: If the variable is set to `false`, the `ilo-virtual-media` boot interface uses swift containers to host the intermediate floppy image and the boot ISO. If the variable is set to `true`, it uses the local web server for hosting the intermediate files. The default value for `use_web_server_for_images` is False.

`http_url`: The value for this variable is prefixed with the generated intermediate files to generate a URL which is attached in the virtual media.

`http_root`: It is the directory location to which ironic conductor copies the intermediate floppy image and the boot ISO.

Note: HTTPS is strongly recommended over HTTP web server configuration for security enhancement. The `ilo-virtual-media` boot interface will send the instances configdrive over an encrypted channel if web server is HTTPS enabled. However for `ilo-uefi-https` boot interface HTTPS webserver is mandatory as this interface only supports HTTPS URLs.

Enable driver

1. Build a deploy ISO (and kernel and ramdisk) image, see *Building or downloading a deploy ramdisk image*
2. See *Glance Configuration* for configuring glance image service with its storage backend as `swift`.
3. Upload this image to Glance:

```
glance image-create --name deploy-ramdisk.iso --disk-format iso --
↪container-format bare < deploy-ramdisk.iso
```

4. Enable hardware type and hardware interfaces in `/etc/ironic/ironic.conf`:

```
[DEFAULT]
enabled_hardware_types = ilo
enabled_bios_interfaces = ilo
enabled_boot_interfaces = ilo-virtual-media,ilo-pxe,ilo-ipxe
enabled_power_interfaces = ilo
enabled_console_interfaces = ilo
enabled_raid_interfaces = agent
enabled_management_interfaces = ilo
enabled_inspect_interfaces = ilo
enabled_rescue_interfaces = agent
```

5. Restart the ironic conductor service:

```
$ service ironic-conductor restart
```

Optional functionalities for the `ilo` hardware type

Boot mode support

The hardware type `ilo` supports automatic detection and setting of boot mode (Legacy BIOS or UEFI).

- When boot mode capability is not configured:
  - If config variable `default_boot_mode` in `[ilo]` section of ironic configuration file is set to either bios or uefi, then iLO driver uses that boot mode for provisioning the baremetal ProLiant servers.
  - If the pending boot mode is set on the node then iLO driver uses that boot mode for provisioning the baremetal ProLiant servers.
  - If the pending boot mode is not set on the node then iLO driver uses uefi boot mode for UEFI capable servers and bios when UEFI is not supported.

- When boot mode capability is configured, the driver sets the pending boot mode to the configured value.
- Only one boot mode (either `uefi` or `bios`) can be configured for the node.
- If the operator wants a node to boot always in `uefi` mode or `bios` mode, then they may use `capabilities` parameter within `properties` field of an ironic node.

  To configure a node in `uefi` mode, then set `capabilities` as below:

```
baremetal node set <node> --property capabilities='boot_mode:uefi'
```

  Nodes having `boot_mode` set to `uefi` may be requested by adding an `extra_spec` to the nova flavor:

```
nova flavor-key ironic-test-3 set capabilities:boot_mode="uefi"
nova boot --flavor ironic-test-3 --image test-image instance-1
```

  If `capabilities` is used in `extra_spec` as above, nova scheduler (`ComputeCapabilitiesFilter`) will match only ironic nodes which have the `boot_mode` set appropriately in `properties/capabilities`. It will filter out rest of the nodes.

  The above facility for matching in nova can be used in heterogeneous environments where there is a mix of `uefi` and `bios` machines, and operator wants to provide a choice to the user regarding boot modes. If the flavor doesnt contain `boot_mode` then nova scheduler will not consider boot mode as a placement criteria, hence user may get either a BIOS or UEFI machine that matches with user specified flavors.

The automatic boot ISO creation for UEFI boot mode has been enabled in Kilo. The manual creation of boot ISO for UEFI boot mode is also supported. For the latter, the boot ISO for the deploy image needs to be built separately and the deploy images `boot_iso` property in glance should contain the glance UUID of the boot ISO. For building boot ISO, add `iso` element to the diskimage-builder command to build the image. For example:

```
disk-image-create ubuntu baremetal iso
```

UEFI Secure Boot Support

The hardware type `ilo` supports secure boot deploy, see *UEFI secure boot mode* for details.

iLO specific notes:

In UEFI secure boot, digitally signed bootloader should be able to validate digital signatures of kernel during boot process. This requires that the bootloader contains the digital signatures of the kernel. For the `ilo-virtual-media` boot interface, it is recommended that `boot_iso` property for user image contains the glance UUID of the boot ISO. If `boot_iso` property is not updated in glance for the user image, it would create the `boot_iso` using bootloader from the deploy iso. This `boot_iso` will be able to boot the user image in UEFI secure boot environment only if the bootloader is signed and can validate digital signatures of user image kernel.

For HPE ProLiant Gen9 servers, one can enroll public key using iLO System Utilities UI. Please refer to section `Accessing Secure Boot options` in HP UEFI System Utilities User Guide. One can also refer to white paper on Secure Boot for Linux on HP ProLiant servers for additional details.

For more up-to-date information, refer iLO driver wiki page

Node Cleaning Support

The hardware type `ilo` and `ilo5` supports node cleaning.

For more information on node cleaning, see *Node cleaning*

Supported Automated Cleaning Operations

- The automated cleaning operations supported are:
  - `reset_bios_to_default`: Resets system ROM settings to default. By default, enabled with priority 10. This clean step is supported only on Gen9 and above servers.
  - `reset_secure_boot_keys_to_default`: Resets secure boot keys to manufacturers defaults. This step is supported only on Gen9 and above servers. By default, enabled with priority 20 .
  - `reset_ilo_credential`: Resets the iLO password, if `ilo_change_password` is specified as part of nodes driver_info. By default, enabled with priority 30.
  - `clear_secure_boot_keys`: Clears all secure boot keys. This step is supported only on Gen9 and above servers. By default, this step is disabled.
  - `reset_ilo`: Resets the iLO. By default, this step is disabled.
  - `erase_devices`: An inband clean step that performs disk erase on all the disks including the disks visible to OS as well as the raw disks visible to Smart Storage Administrator (SSA). This step supports erasing of the raw disks visible to SSA in Proliant servers only with the ramdisk created using diskimage-builder from Ocata release. By default, this step is disabled. See *Disk Erase Support* for more details.
- For supported in-band cleaning operations, see *In-band vs out-of-band*.
- All the automated cleaning steps have an explicit configuration option for priority. In order to disable or change the priority of the automated clean steps, respective configuration option for priority should be updated in ironic.conf.
- Updating clean step priority to 0, will disable that particular clean step and will not run during automated cleaning.
- Configuration Options for the automated clean steps are listed under `[ilo]` and `[deploy]` section in ironic.conf

```
[ilo]
clean_priority_reset_ilo=0
clean_priority_reset_bios_to_default=10
clean_priority_reset_secure_boot_keys_to_default=20
clean_priority_clear_secure_boot_keys=0
clean_priority_reset_ilo_credential=30

[deploy]
erase_devices_priority=0
```

For more information on node automated cleaning, see *Automated cleaning*

Supported Manual Cleaning Operations

- The manual cleaning operations supported are:

  activate_license: Activates the iLO Advanced license. This is an out-of-band manual cleaning step associated with the `management` interface. See *Activating iLO Advanced license as manual clean step* for user guidance on usage. Please note that this operation cannot be performed using the `ilo-virtual-media` boot interface as it needs this type of advanced license already active to use virtual media to boot into to start cleaning operation. Virtual media is an advanced feature. If an advanced license is already active and the user wants to overwrite the current license key, for example in case of a multi-server activation key delivered with a flexible-quantity kit or after completing an Activation Key Agreement (AKA), then the driver can still be used for executing this cleaning step.

  clear_ca_certificates: Removes the CA certificates from iLO. See *Removing CA certificates from iLO as manual clean step* for user guidance on usage.

  apply_configuration: Applies given BIOS settings on the node. See *BIOS configuration support*. This step is part of the `bios` interface.

  factory_reset: Resets the BIOS settings on the node to factory defaults. See *BIOS configuration support*. This step is part of the `bios` interface.

  create_configuration: Applies RAID configuration on the node. See *RAID Configuration* for more information. This step is part of the `raid` interface.

  delete_configuration: Deletes RAID configuration on the node. See *RAID Configuration* for more information. This step is part of the `raid` interface.

  update_firmware: Updates the firmware of the devices. Also an out-of-band step associated with the `management` interface. See *Initiating firmware update as manual clean step* for user guidance on usage. The supported devices for firmware update are: `ilo`, `cpld`, `power_pic`, `bios` and `chassis`. Please refer to below table for their commonly used descriptions.

Device	Description
`ilo`	BMC for HPE ProLiant servers
`cpld`	System programmable logic device
`power_pic`	Power management controller
`bios`	HPE ProLiant System ROM
`chassis`	System chassis device

  Some devices firmware cannot be updated via this method, such as: storage controllers, host bus adapters, disk drive firmware, network interfaces and Onboard Administrator (OA).

  update_firmware_sum: Updates all or list of user specified firmware components on the node using Smart Update Manager (SUM). It is an inband step associated with the `management` interface. See *Smart Update Manager (SUM) based firmware update* for more information on usage.

  security_parameters_update: Updates the Security Parameters. See *Updating security parameters as manual clean step* for user guidance on usage. The supported security parameters for this clean step are: `Password_Complexity`, `RequiredLoginForiLORBSU`, `IPMI/DCMI_Over_LAN`, `RequireHostAuthentication` and `Secure_Boot`.

update_minimum_password_length: Updates the Minimum Password Length security parameter. See *Update Minimum Password Length security parameter as manual clean step* for user guidance on usage.

update_auth_failure_logging_threshold: Updates the Authentication Failure Logging security parameter. See *Update Authentication Failure Logging security parameter as manual clean step* for user guidance on usage.

- iLO with firmware version 1.5 is minimally required to support all the operations.

For more information on node manual cleaning, see *Manual cleaning*

Node Deployment Customization

The hardware type `ilo` and `ilo5` supports customization of node deployment via deploy templates, see *Node Deployment*.

The supported deploy steps are:

- **apply_configuration:** Applies given BIOS settings on the node. See *BIOS configuration support*. This step is part of the `bios` interface.
- **factory_reset:** Resets the BIOS settings on the node to factory defaults. See *BIOS configuration support*. This step is part of the `bios` interface.
- **reset_bios_to_default:** Resets system ROM settings to default. This step is supported only on Gen9 and above servers. This step is part of the `management` interface.
- **reset_secure_boot_keys_to_default:** Resets secure boot keys to manufacturers defaults. This step is supported only on Gen9 and above servers. This step is part of the `management` interface.
- **reset_ilo_credential:** Resets the iLO password. The password need to be specified in `ilo_password` argument of the step. This step is part of the `management` interface.
- **clear_secure_boot_keys:** Clears all secure boot keys. This step is supported only on Gen9 and above servers. This step is part of the `management` interface.
- **reset_ilo:** Resets the iLO. This step is part of the `management` interface.
- **update_firmware:** Updates the firmware of the devices. This step is part of the `management` interface. See *Initiating firmware update as manual clean step* for user guidance on usage. The supported devices for firmware update are: `ilo`, `cpld`, `power_pic`, `bios` and `chassis`. This step is part of `management` interface. Please refer to below table for their commonly used descriptions.

Device	Description
`ilo`	BMC for HPE ProLiant servers
`cpld`	System programmable logic device
`power_pic`	Power management controller
`bios`	HPE ProLiant System ROM
`chassis`	System chassis device

Some devices firmware cannot be updated via this method, such as: storage controllers, host bus adapters, disk drive firmware, network interfaces and Onboard Administrator (OA).

- **flash_firmware_sum:** Updates all or list of user specified firmware components on the node using Smart Update Manager (SUM). It is an inband step associated with the `management` interface. See *Smart Update Manager (SUM) based firmware update* for more information on usage.
- **apply_configuration:** Applies RAID configuration on the node. See *RAID Configuration* for more information. This step is part of the `raid` interface.

Example of using deploy template with the Compute service

Create a deploy template with a single step:

```
baremetal deploy template create \
    CUSTOM_HYPERTHREADING_ON \
    --steps '[{"interface": "bios", "step": "apply_configuration", "args": {
↪"settings": [{"name": "ProcHyperthreading", "value": "Enabled"}]}, "priority
↪": 150}]'
```

Add the trait `CUSTOM_HYPERTHREADING_ON` to the node represented by `$node_ident`:

```
baremetal node add trait $node_ident CUSTOM_HYPERTHREADING_ON
```

Update the flavor `bm-hyperthreading-on` in the Compute service with the following property:

```
openstack flavor set --property trait:CUSTOM_HYPERTHREADING_ON=required bm-
↪hyperthreading-on
```

Creating a Compute instance with this flavor will ensure that the instance is scheduled only to Bare Metal nodes with the `CUSTOM_HYPERTHREADING_ON` trait. When an instance is created using the `bm-hyperthreading-on` flavor, then the deploy steps of deploy template `CUSTOM_HYPERTHREADING_ON` will be executed during the deployment of the scheduled node, causing Hyperthreading to be enabled in the nodes BIOS configuration.

Hardware Inspection Support

The hardware type `ilo` supports hardware inspection.

Note:

- The disk size is returned by RIBCL/RIS only when RAID is preconfigured on the storage. If the storage is Direct Attached Storage, then RIBCL/RIS fails to get the disk size.
- The SNMPv3 inspection gets disk size for all types of storages. If RIBCL/RIS is unable to get disk size and SNMPv3 inspection is requested, the proliantutils does SNMPv3 inspection to get the disk size. If proliantutils is unable to get the disk size, it raises an error. This feature is available in proliantutils release version >= 2.2.0.
- The iLO must be updated with SNMPv3 authentication details. Pleae refer to the section *SNMPv3 Authentication* in HPE iLO4 User Guide for setting up authentication details on iLO. The following parameters are mandatory to be given in driver_info for SNMPv3 inspection:
  - `snmp_auth_user` : The SNMPv3 user.

  - snmp_auth_prot_password : The auth protocol pass phrase.
  - snmp_auth_priv_password : The privacy protocol pass phrase.

  The following parameters are optional for SNMPv3 inspection:

  - snmp_auth_protocol : The Auth Protocol. The valid values are MD5 and SHA. The iLO default value is MD5.
  - snmp_auth_priv_protocol : The Privacy protocol. The valid values are AES and DES. The iLO default value is DES.

The inspection process will discover the following essential properties (properties required for scheduling deployment):

- memory_mb: memory size
- cpus: number of cpus
- cpu_arch: cpu architecture
- local_gb: disk size

Inspection can also discover the following extra capabilities for iLO driver:

- ilo_firmware_version: iLO firmware version
- rom_firmware_version: ROM firmware version
- secure_boot: secure boot is supported or not. The possible values are true or false. The value is returned as true if secure boot is supported by the server.
- server_model: server model
- pci_gpu_devices: number of gpu devices connected to the bare metal.
- nic_capacity: the max speed of the embedded NIC adapter.
- sriov_enabled: true, if server has the SRIOV supporting NIC.
- has_rotational: true, if server has HDD disk.
- has_ssd: true, if server has SSD disk.
- has_nvme_ssd: true, if server has NVME SSD disk.
- cpu_vt: true, if server supports cpu virtualization.
- hardware_supports_raid: true, if RAID can be configured on the server using RAID controller.
- nvdimm_n: true, if server has NVDIMM_N type of persistent memory.
- persistent_memory: true, if server has persistent memory.
- logical_nvdimm_n: true, if server has logical NVDIMM_N configured.
- rotational_drive_<speed>_rpm: The capabilities rotational_drive_4800_rpm, rotational_drive_5400_rpm, rotational_drive_7200_rpm, rotational_drive_10000_rpm and rotational_drive_15000_rpm are set to true if the server has HDD drives with speed of 4800, 5400, 7200, 10000 and 15000 rpm respectively.
- logical_raid_level_<raid_level>: The capabilities logical_raid_level_0, logical_raid_level_1, logical_raid_level_2, logical_raid_level_5, logical_raid_level_6, logical_raid_level_10, logical_raid_level_50 and

`logical_raid_level_60` are set to true if any of the raid levels among 0, 1, 2, 5, 6, 10, 50 and 60 are configured on the system.

- `overall_security_status`: `Ok` or `Risk` or `Ignored` as returned by iLO security dashboard. iLO computes the overall security status by evaluating the security status for each of the security parameters. Admin needs to fix the actual parameters and then re-inspect so that iLO can recompute the overall security status. If the all security params, whose `security_status` is `Risk`, have the `Ignore` field set to `True`, then iLO sets the overall security status value as `Ignored`. All the security params must have the `security_status` as `Ok` for the `overall_security_status` to have the value as `Ok`.
- `last_firmware_scan_status`: `Ok` or `Risk` as returned by iLO security dashboard. This denotes security status of the last firmware scan done on the system. If it is `Risk`, the recommendation is to run clean_step `update_firmware_sum` without any specific firmware components so that firmware is updated for all the components using latest SPP (Service Provider Pack) ISO and then re-inspect to get the security status again.
- `security_override_switch`: `Ok` or `Risk` as returned by iLO security dashboard. This is disable/enable login to the iLO using credentials. This can be toggled only by physical visit to the bare metal.
- `gpu_<vendor>_count`: Integer value. The capability name is dynamically formed as gpu_<vendor>_count. The vendor name is replaced in the <vendor>. If the vendor name is not returned by the hardware, then vendor ID in hexadecimal form is replaced in the capability name. Examples: {gpu_Nvidia_count: 1}, {gpu_0x102b_count: 1}.
- `gpu_<vendor_device_name>_count`: Integer value. The capability name is formed dynamically by replacing the gpu device name as returned by ilo in <vendor_device_name>. Examples: {gpu_Nvidia_Tesla_M10_count: 1}, {gpu_Embedded_Video_Controller_count: 1}
- `gpu_<vendor_device_name>`: Boolean. The capability name is formed dynamically by replacing the gpu device name as returned by ilo in <vendor_device_name>. Examples: {gpu_Nvidia_Tesla_M10: True}, {gpu_Embedded_Video_Controller: True}

Note:

- The capability `nic_capacity` can only be discovered if ipmitool version >= 1.8.15 is used on the conductor. The latest version can be downloaded from here.
- The iLO firmware version needs to be 2.10 or above for nic_capacity to be discovered.
- To discover IPMI based attributes you need to enable iLO feature IPMI/DCMI over LAN Access on iLO4 and iLO5 management engine.
- The proliantutils returns only active NICs for Gen10 ProLiant HPE servers. The user would need to delete the ironic ports corresponding to inactive NICs for Gen8 and Gen9 servers as proliantutils returns all the discovered (active and otherwise) NICs for Gen8 and Gen9 servers and ironic ports are created for all of them. Inspection logs a warning if the node under inspection is Gen8 or Gen9.
- The security dashboard capabilities are applicable only for Gen10 ProLiant HPE servers and above. To fix the security dashboard parameters value from `Risk` to `Ok`, user need to fix the parameters separately and re-inspect to see the security status of the parameters.

The operator can specify these capabilities in nova flavor for node to be selected for scheduling:

```
nova flavor-key my-baremetal-flavor set capabilities:server_model="<in> Gen8"

nova flavor-key my-baremetal-flavor set capabilities:nic_capacity="10Gb"

nova flavor-key my-baremetal-flavor set capabilities:ilo_firmware_version="
↪<in> 2.10"

nova flavor-key my-baremetal-flavor set capabilities:has_ssd="true"
```

See *Capabilities discovery* for more details and examples.

Swiftless deploy for intermediate images

The hardware type `ilo` with `ilo-virtual-media` as boot interface can deploy and boot the server with and without `swift` being used for hosting the intermediate temporary floppy image (holding metadata for deploy kernel and ramdisk) and the boot ISO. A local HTTP(S) web server on each conductor node needs to be configured. Please refer to *Web server configuration on conductor* for more information. The HTTPS web server needs to be enabled (instead of HTTP web server) in order to send management information and images in encrypted channel over HTTPS.

Note: This feature assumes that the user inputs are on Glance which uses swift as backend. If swift dependency has to be eliminated, please refer to *HTTP(S) Based Deploy Support* also.

Deploy Process

Please refer to *Netboot in swiftless deploy for intermediate images* for partition image support and *Localboot in swiftless deploy for intermediate images* for whole disk image support.

HTTP(S) Based Deploy Support

The user input for the images given in `driver_info` like `deploy_iso`, `deploy_kernel` and `deploy_ramdisk` and in `instance_info` like `image_source`, `kernel`, `ramdisk` and `boot_iso` may also be given as HTTP(S) URLs.

The HTTP(S) web server can be configured in many ways. For the Apache web server on Ubuntu, refer here. The web server may reside on a different system than the conductor nodes, but its URL must be reachable by the conductor and the bare metal nodes.

Deploy Process

Please refer to *Netboot with HTTP(S) based deploy* for partition image boot and *Localboot with HTTP(S) based deploy* for whole disk image boot.

Support for iLO driver with Standalone Ironic

It is possible to use ironic as standalone services without other OpenStack services. The `ilo` hardware type can be used in standalone ironic. This feature is referred to as `iLO driver with standalone ironic` in this document.

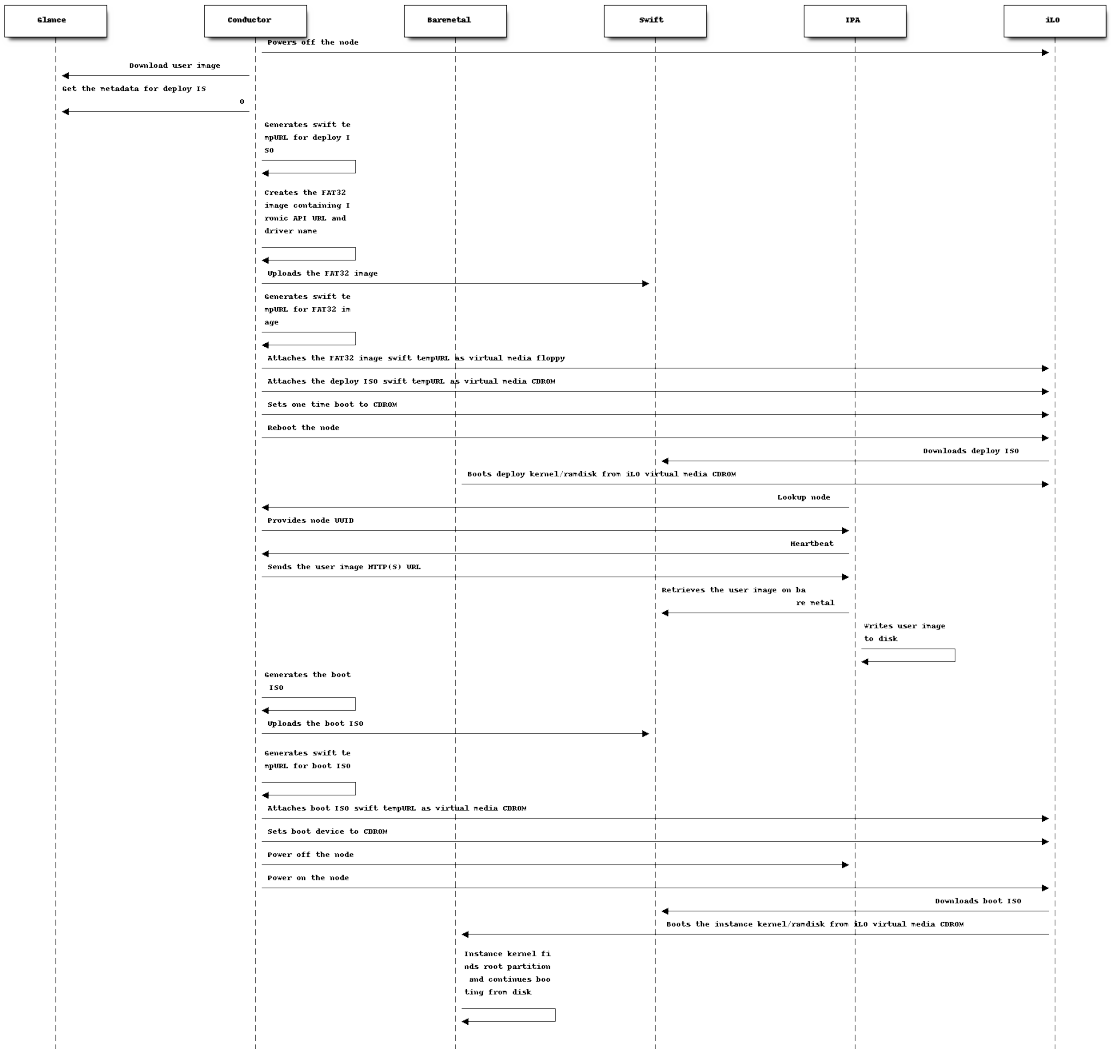
Configuration

The HTTP(S) web server needs to be configured as described in *HTTP(S) Based Deploy Support* and *Web server configuration on conductor* needs to be configured for hosting intermediate images on conductor as described in *Swiftless deploy for intermediate images*.

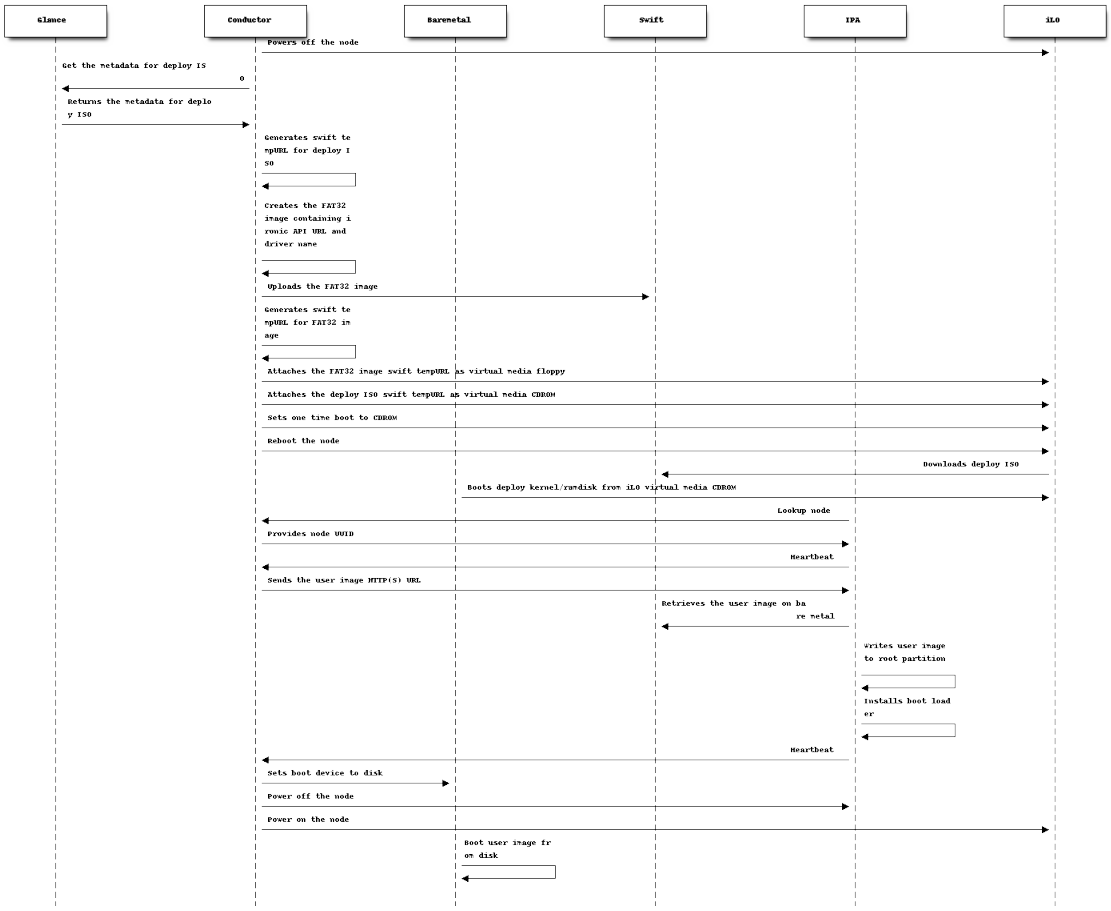
Deploy Process

Note: Network boot is deprecated and will be removed in the Zed release.

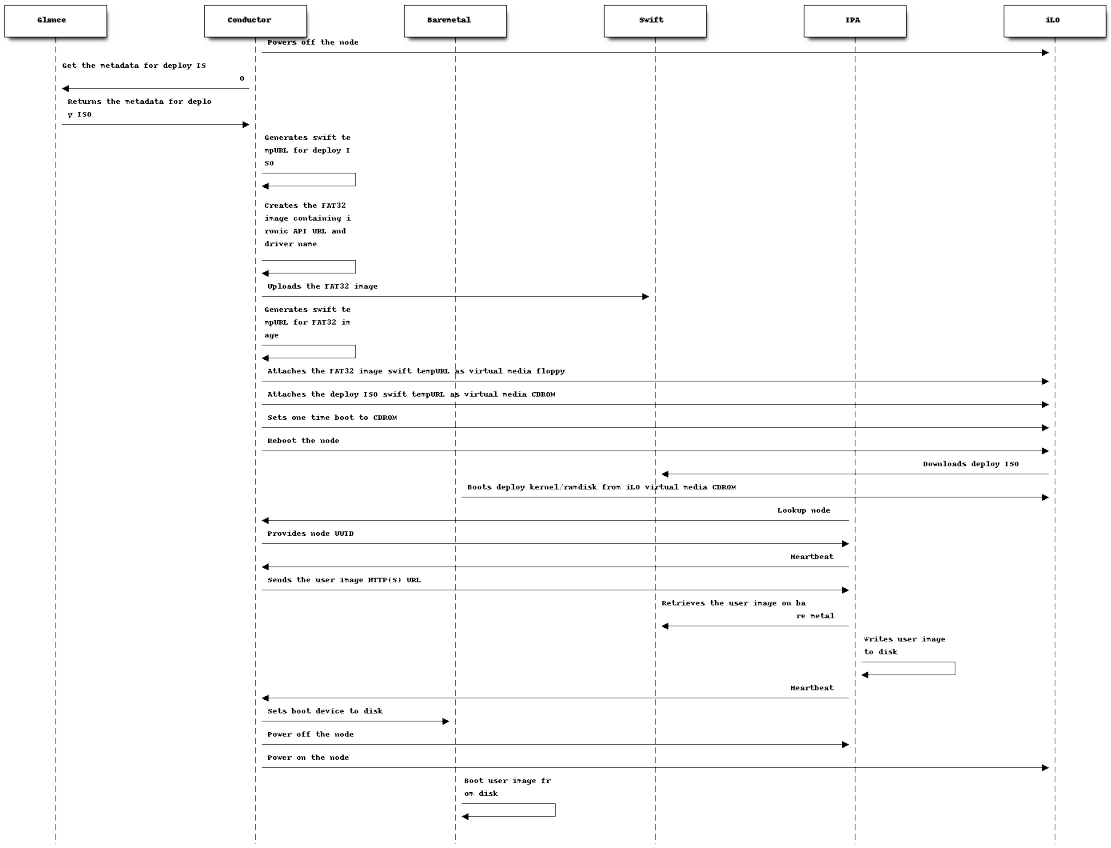
Netboot with glance and swift



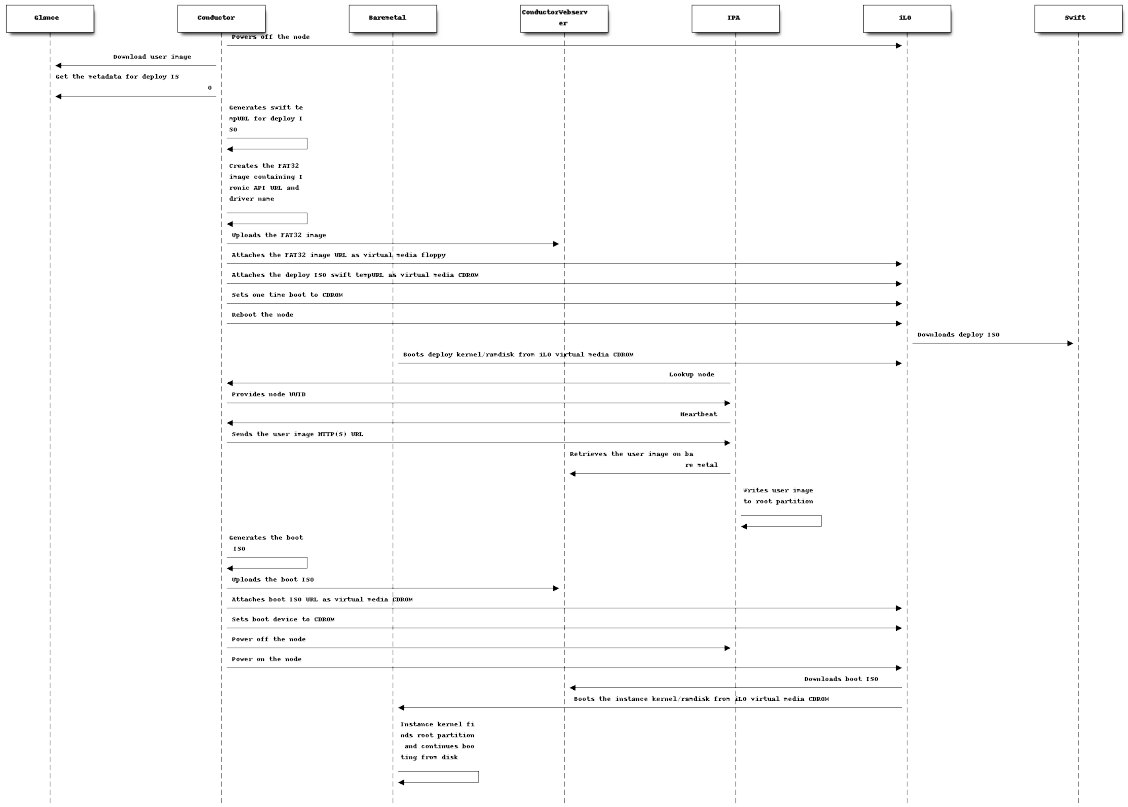
Localboot with glance and swift for partition images



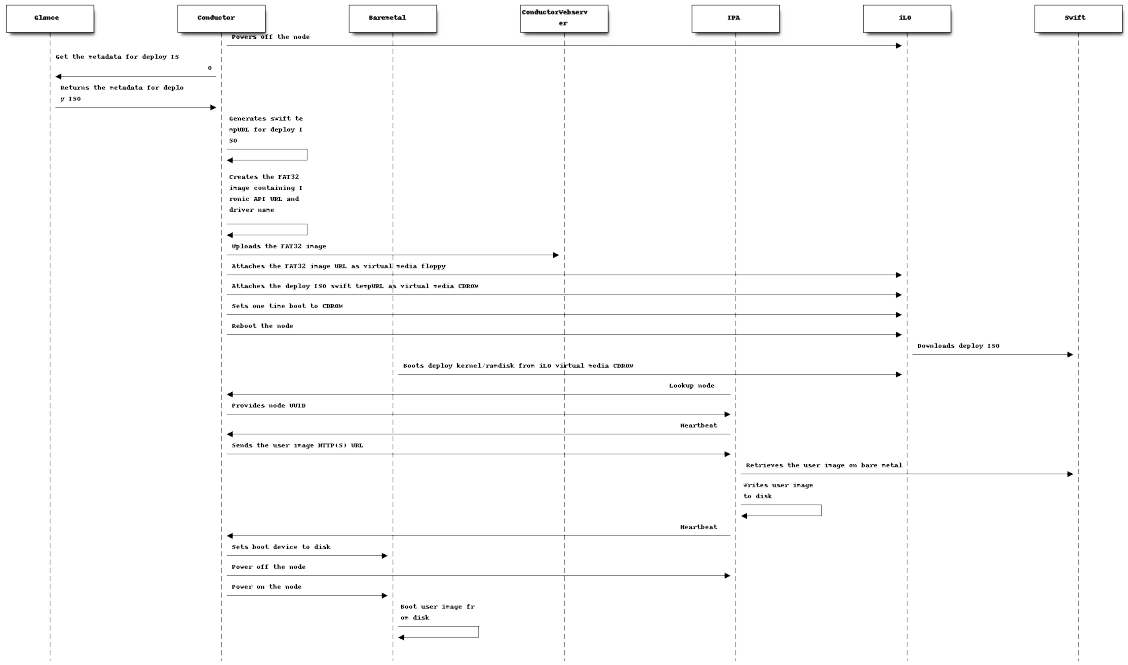
Localboot with glance and swift



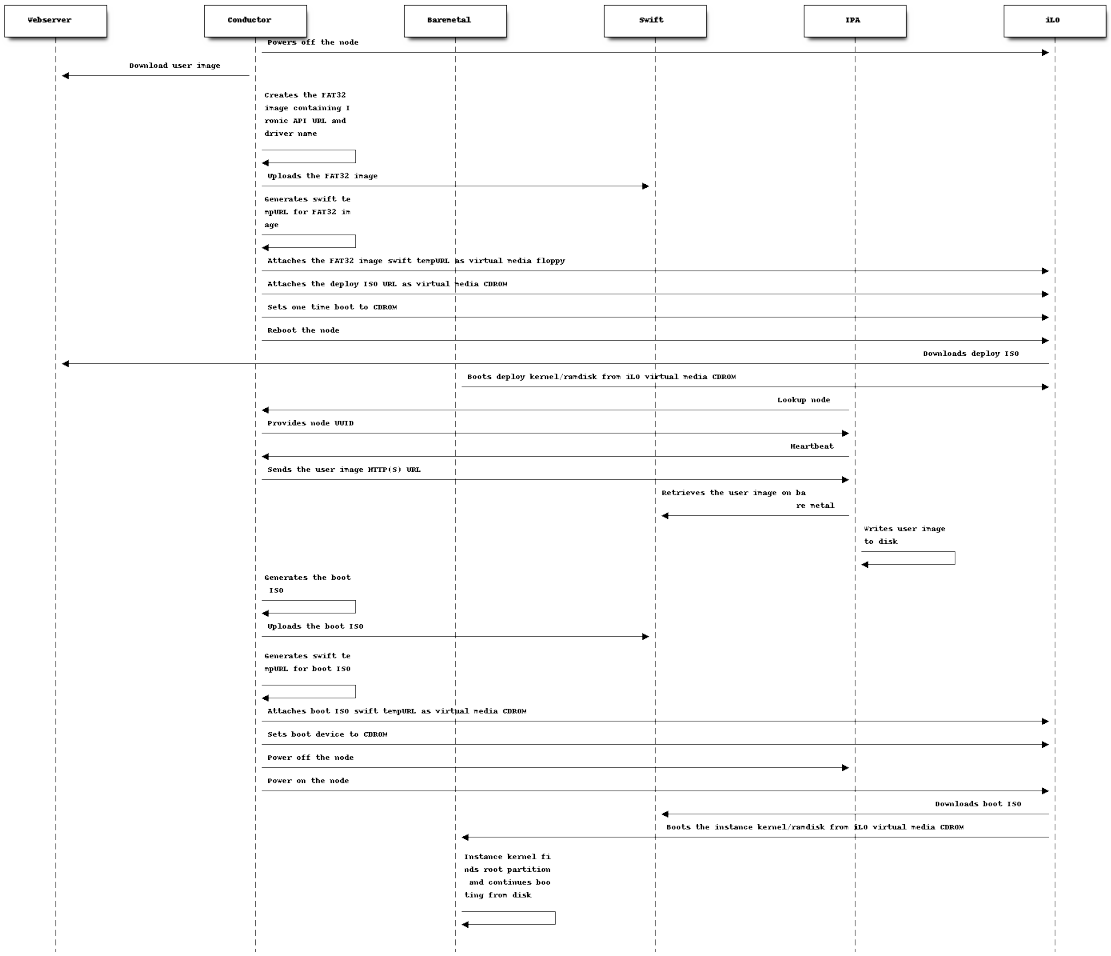
Netboot in swiftless deploy for intermediate images



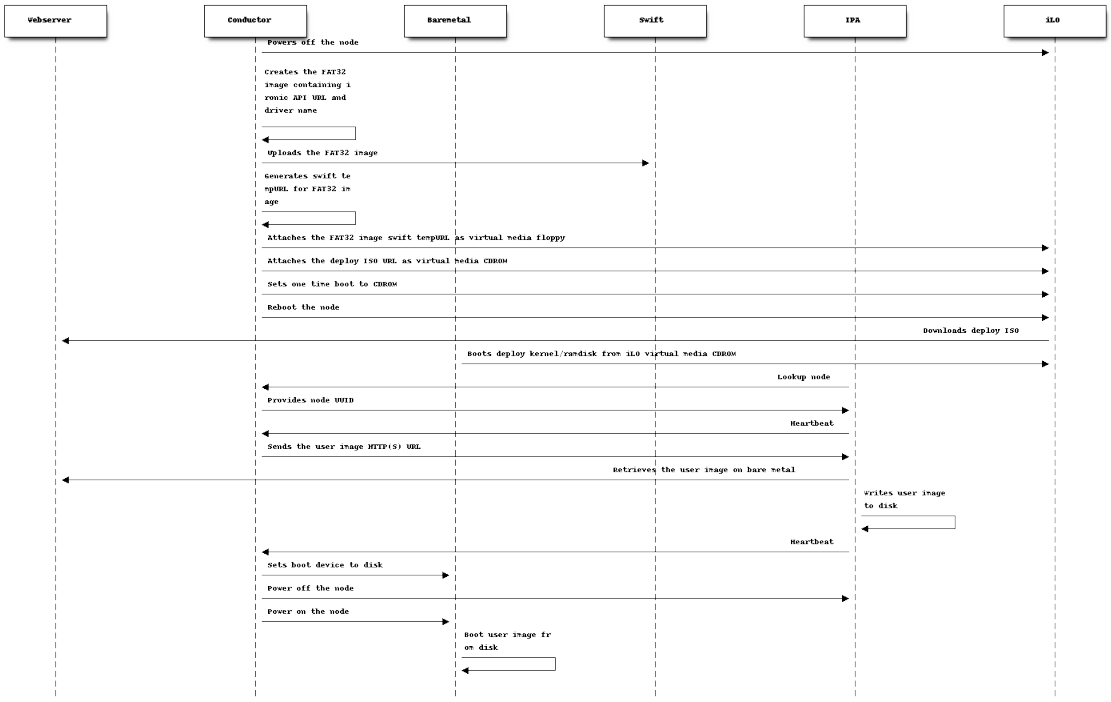
Localboot in swiftless deploy for intermediate images



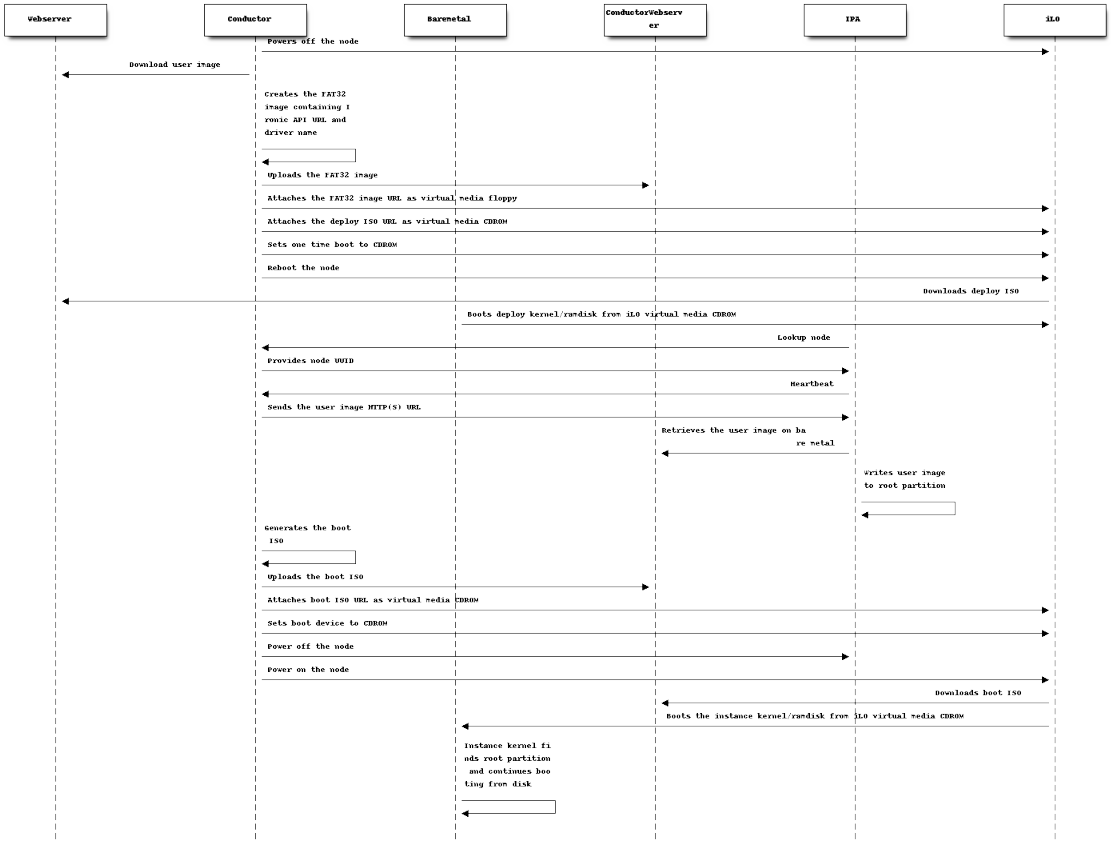
Netboot with HTTP(S) based deploy



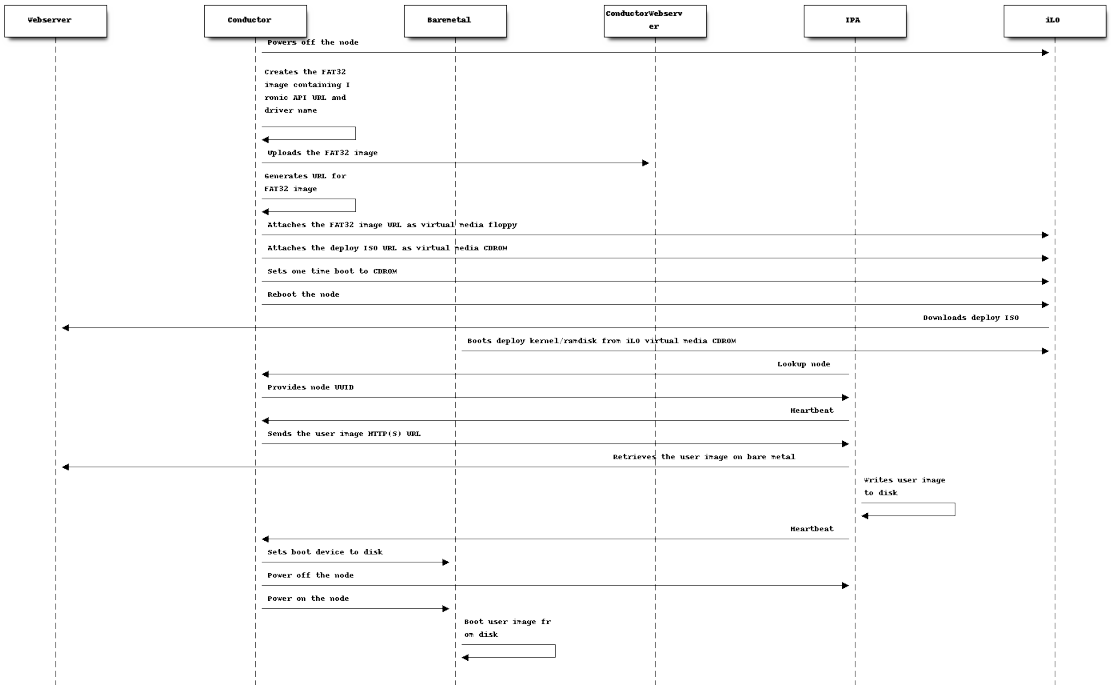
Localboot with HTTP(S) based deploy



Netboot in standalone ironic



Localboot in standalone ironic



Activating iLO Advanced license as manual clean step

iLO driver can activate the iLO Advanced license key as a manual cleaning step. Any manual cleaning step can only be initiated when a node is in the `manageable` state. Once the manual cleaning is finished, the node will be put in the `manageable` state again. User can follow steps from *Manual cleaning* to initiate manual cleaning operation on a node.

An example of a manual clean step with `activate_license` as the only clean step could be:

```
"clean_steps": [{
    "interface": "management",
    "step": "activate_license",
    "args": {
        "ilo_license_key": "ABC12-XXXXX-XXXXX-XXXXX-YZ345"
    }
}]
```

The different attributes of `activate_license` clean step are as follows:

Attribute	Description
`interface`	Interface of clean step, here `management`
`step`	Name of clean step, here `activate_license`
`args`	Keyword-argument entry (<name>: <value>) being passed to clean step
`args.ilo_license_key`	iLO Advanced license key to activate enterprise features. This is mandatory.

Removing CA certificates from iLO as manual clean step

iLO driver can remove the invalidated CA certificates as a manual step. Any manual cleaning step can only be initiated when a node is in the `manageable` state. Once the manual cleaning is finished, the node will be put in the `manageable` state again. User can follow steps from *Manual cleaning* to initiate manual cleaning operation on a node.

An example of a manual clean step with `clear_ca_certificates` as the only clean step could be:

```
"clean_steps": [{
    "interface": "management",
    "step": "clear_ca_certificates",
    "args": {
        "certificate_files" : ["/path/to/certsA", "/path/to/certsB"]
    }
}]
```

The different attributes of `clear_ca_certificates` clean step are as follows:

Attribute	Description
`interface`	Interface of clean step, here `management`
`step`	Name of clean step, here `clear_ca_certificates`
`args`	Keyword-argument entry (<name>: <value>) being passed to clean step
`args.certificate_files` This is mandatory.	List of CA certificates which are to be removed.

Initiating firmware update as manual clean step

iLO driver can invoke secure firmware update as a manual cleaning step. Any manual cleaning step can only be initiated when a node is in the `manageable` state. Once the manual cleaning is finished, the node will be put in the `manageable` state again. A user can follow steps from *Manual cleaning* to initiate manual cleaning operation on a node.

An example of a manual clean step with `update_firmware` as the only clean step could be:

```
"clean_steps": [{
    "interface": "management",
    "step": "update_firmware",
    "args": {
        "firmware_update_mode": "ilo",
        "firmware_images":[
            {
                "url": "file:///firmware_images/ilo/1.5/CP024444.scexe",
                "checksum": "a94e683ea16d9ae44768f0a65942234d",
                "component": "ilo"
            },
            {
                "url": "swift://firmware_container/cpld2.3.rpm",
```

(continues on next page)

(continued from previous page)

```
                "checksum": "<md5-checksum-of-this-file>",
                "component": "cpld"
            },
            {
                "url": "http://my_address:port/firmwares/bios_vLatest.scexe",
                "checksum": "<md5-checksum-of-this-file>",
                "component": "bios"
            },
            {
                "url": "https://my_secure_address_url/firmwares/chassis_
↪vLatest.scexe",
                "checksum": "<md5-checksum-of-this-file>",
                "component": "chassis"
            },
            {
                "url": "file:///home/ubuntu/firmware_images/power_pic/pmc_v3.
↪0.bin",
                "checksum": "<md5-checksum-of-this-file>",
                "component": "power_pic"
            }
        ]
    }
}]
```

The different attributes of `update_firmware` clean step are as follows:

Attribute	Description
`interface`	Interface of clean step, here `management`
`step`	Name of clean step, here `update_firmware`
`args`	Keyword-argument entry (<name>: <value>) being passed to clean step
`args.firmware_update_mode`	Mode (or mechanism) of out-of-band firmware update. Supported value is `ilo`. This is mandatory.
`args.firmware_images`	Ordered list of dictionaries of images to be flashed. This is mandatory.

Each firmware image block is represented by a dictionary (JSON), in the form:

```
{
  "url": "<url of firmware image file>",
  "checksum": "<md5 checksum of firmware image file to verify the image>",
  "component": "<device on which firmware image will be flashed>"
}
```

All the fields in the firmware image block are mandatory.

- The different types of firmware url schemes supported are: `file`, `http`, `https` and `swift`.

> **Note:** This feature assumes that while using `file` url scheme the file path is on the conductor controlling the node.

Note: The `swift` url scheme assumes the swift account of the `service` project. The `service` project (tenant) is a special project created in the Keystone system designed for the use of the core OpenStack services. When Ironic makes use of Swift for storage purpose, the account is generally `service` and the container is generally `ironic` and `ilo` driver uses a container named `ironic_ilo_container` for their own purpose.

Note: While using firmware files with a `.rpm` extension, make sure the commands `rpm2cpio` and `cpio` are present on the conductor, as they are utilized to extract the firmware image from the package.

- The firmware components that can be updated are: `ilo`, `cpld`, `power_pic`, `bios` and `chassis`.
- The firmware images will be updated in the order given by the operator. If there is any error during processing of any of the given firmware images provided in the list, none of the firmware updates will occur. The processing error could happen during image download, image checksum verification or image extraction. The logic is to process each of the firmware files and update them on the devices only if all the files are processed successfully. If, during the update (uploading and flashing) process, an update fails, then the remaining updates, if any, in the list will be aborted. But it is recommended to triage and fix the failure and re-attempt the manual clean step `update_firmware` for the aborted `firmware_images`.

  The devices for which the firmwares have been updated successfully would start functioning using their newly updated firmware.
- As a troubleshooting guidance on the complete process, check Ironic conductor logs carefully to see if there are any firmware processing or update related errors which may help in root causing or gain an understanding of where things were left off or where things failed. You can then fix or work around and then try again. A common cause of update failure is HPE Secure Digital Signature check failure for the firmware image file.
- To compute `md5` checksum for your image file, you can use the following command:

```
$ md5sum image.rpm
66cdb090c80b71daa21a67f06ecd3f33  image.rpm
```

Smart Update Manager (SUM) based firmware update

The firmware update based on SUM is an inband clean/deploy step supported by iLO driver. The firmware update is performed on all or list of user specified firmware components on the node. Refer to SUM User Guide to get more information on SUM based firmware update.

Note: `update_firmware_sum` clean step requires the agent ramdisk with `Proliant Hardware Manager` from the proliantutils version 2.5.0 or higher. See *DIB support for Proliant Hardware Manager* to create the agent ramdisk with `Proliant Hardware Manager`.

Note: `flash_firmware_sum` deploy step requires the agent ramdisk with `Proliant Hardware Manager` from the proliantutils version 2.9.5 or higher. See *DIB support for Proliant Hardware Manager*

to create the agent ramdisk with `Proliant Hardware Manager`.

The attributes of `update_firmware_sum/flash_firmware_sum` step are as follows:

Attribute	Description
`interface`	Interface of the clean step, here `management`
`step`	Name of the clean step, here `update_firmware_sum`
`args`	Keyword-argument entry (<name>: <value>) being passed to the clean step

The keyword arguments used for the step are as follows:

- `url`: URL of SPP (Service Pack for Proliant) ISO. It is mandatory. The URL schemes supported are `http`, `https` and `swift`.
- `checksum`: MD5 checksum of SPP ISO to verify the image. It is mandatory.
- `components`: List of filenames of the firmware components to be flashed. It is optional. If not provided, the firmware update is performed on all the firmware components.

The step performs an update on all or a list of firmware components and returns the SUM log files. The log files include `hpsum_log.txt` and `hpsum_detail_log.txt` which holds the information about firmware components, firmware version for each component and their update status. The log object will be named with the following pattern:

```
<node>[_<instance-uuid>]_update_firmware_sum_<timestamp yyyy-mm-dd-hh-mm-ss>.
↪tar.gz
or
<node>[_<instance-uuid>]_flash_firmware_sum_<timestamp yyyy-mm-dd-hh-mm-ss>.
↪tar.gz
```

Refer to *Retrieving logs from the deploy ramdisk* for more information on enabling and viewing the logs returned from the ramdisk.

An example of `update_firmware_sum` clean step:

```
{
    "interface": "management",
    "step": "update_firmware_sum",
    "args":
        {
            "url": "http://my_address:port/SPP.iso",
            "checksum": "abcdefxyz",
            "components": ["CP024356.scexe", "CP008097.exe"]
        }
}
```

The step fails if there is any error in the processing of step arguments. The processing error could happen during validation of components file extension, image download, image checksum verification or image extraction. In case of a failure, check Ironic conductor logs carefully to see if there are any validation or firmware processing related errors which may help in root cause analysis or gaining an understanding of where things were left off or where things failed. You can then fix or work around and then try again.

Warning: This feature is officially supported only with RHEL and SUSE based IPA ramdisk. Refer to SUM for supported OS versions for specific SUM version.

Note: Refer Guidelines for SPP ISO for steps to get SPP (Service Pack for ProLiant) ISO.

Updating security parameters as manual clean step

iLO driver can invoke security parameters update as a manual clean step. Any manual cleaning step can only be initiated when a node is in the `manageable` state. Once the manual cleaning is finished, the node will be put in the `manageable` state again. A user can follow steps from *Manual cleaning* to initiate manual cleaning operation on a node. This feature is only supported for iLO5 based hardware.

An example of a manual clean step with `security_parameters_update` as the only clean step could be:

```
"clean_steps": [{
    "interface": "management",
    "step": "security_parameters_update",
    "args": {
        "security_parameters":[
            {
                "param": "password_complexity",
                "enable": "True",
                "ignore": "False"
            },
            {
                "param": "require_login_for_ilo_rbsu",
                "enable": "True",
                "ignore": "False"
            },
            {
                "param": "ipmi_over_lan",
                "enable": "True",
                "ignore": "False"
            },
            {
                "param": "secure_boot",
                "enable": "True",
                "ignore": "False"
            },
            {
                "param": "require_host_authentication",
                "enable": "True",
                "ignore": "False"
            }
        ]
    }
```

(continues on next page)

(continued from previous page)

```
}]
```

The different attributes of `security_parameters_update` clean step are as follows:

Attribute	Description
`interface`	Interface of clean step, here `management`
`step`	Name of clean step, here `security_parameters_update`
`args`	Keyword-argument entry (<name>: <value>) being passed to clean step
`args.security_parameters`	Ordered list of dictionaries of security parameters to be updated. This is mandatory.

Each security parameter block is represented by a dictionary (JSON), in the form:

```
{
  "param": "<security parameter name>",
  "enable": "security parameter to be enabled/disabled",
  "ignore": "security parameter status to be ignored or not"
}
```

In all of these fields, `param` field is mandatory. Remaining fields are boolean and are optional. If user doesnt pass any value then for `enable` field the default will be True and for `ignore` field default will be False.

- The Security Parameters which are supported for this clean step are: `Password_Complexity`, `RequiredLoginForiLORBSU`, `RequireHostAuthentication`, `IPMI/DCMI_Over_LAN` and `Secure_Boot`.

Update Minimum Password Length security parameter as manual clean step

iLO driver can invoke `Minimum Password Length` security parameter update as a manual clean step. This feature is only supported for iLO5 based hardware.

An example of a manual clean step with `update_minimum_password_length` as the only clean step could be:

```
"clean_steps": [{
    "interface": "management",
    "step": "update_minimum_password_length",
    "args": {
        "password_length": "8",
        "ignore": "False"
    }
}]
```

Both the arguments `password_length` and `ignore` are optional. The accepted values for password_length are 0 to 39. If user doesnt pass any value, the default value for password_length will be 8 and for ignore the default value be False.

Update Authentication Failure Logging security parameter as manual clean step

iLO driver can invoke `Authentication Failure Logging` security parameter update as a manual clean step. This feature is only supported for iLO5 based hardware.

An example of a manual clean step with `Authentication Failure Logging` as the only clean step could be:

```
"clean_steps": [{
    "interface": "management",
    "step": "update_auth_failure_logging_threshold",
    "args": {
        "logging_threshold": "1",
        "ignore": "False"
    }
}]
```

Both the arguments `logging_threshold` and `ignore` are optional. The accepted values for logging_threshold are 0 to 5. If user doesnt pass any value, the default value for logging_threshold will be 1 and for ignore the default value be False. If user passes the value of logging_threshold as 0, the Authentication Failure Logging security parameter will be disabled.

RAID Support

The inband RAID functionality is supported by iLO driver. See *RAID Configuration* for more information. Bare Metal service update node with following information after successful configuration of RAID:

- Node `properties/local_gb` is set to the size of root volume.
- Node `properties/root_device` is filled with `wwn` details of root volume. It is used by iLO driver as root device hint during provisioning.
- The value of raid level of root volume is added as `raid_level` capability to the nodes `capabilities` parameter within `properties` field. The operator can specify the `raid_level` capability in nova flavor for node to be selected for scheduling:

```
nova flavor-key ironic-test set capabilities:raid_level="1+0"
nova boot --flavor ironic-test --image test-image instance-1
```

DIB support for Proliant Hardware Manager

Install `ironic-python-agent-builder` following the guide[1]

To create an agent ramdisk with `Proliant Hardware Manager`, use the `proliant-tools` element in DIB:

```
ironic-python-agent-builder -o proliant-agent-ramdisk -e proliant-tools fedora
```

[1] *ironic-python-agent-builder*: https://docs.openstack.org/ironic-python-agent-builder/latest/install/index.html

Disk Erase Support

`erase_devices` is an inband clean step supported by iLO driver. It performs erase on all the disks including the disks visible to OS as well as the raw disks visible to the Smart Storage Administrator (SSA).

This inband clean step requires `ssacli` utility starting from version `2.60-19.0` to perform the erase on physical disks. See the ssacli documentation for more information on ssacli utility and different erase methods supported by SSA.

The disk erasure via `shred` is used to erase disks visible to the OS and its implementation is available in Ironic Python Agent. The raw disks connected to the Smart Storage Controller are erased using Sanitize erase which is a ssacli supported erase method. If Sanitize erase is not supported on the Smart Storage Controller the disks are erased using One-pass erase (overwrite with zeros).

This clean step is supported when the agent ramdisk contains the `Proliant Hardware Manager` from the proliantutils version 2.3.0 or higher. This clean step is performed as part of automated cleaning and it is disabled by default. See *In-band vs out-of-band* for more information on enabling/disabling a clean step.

Install `ironic-python-agent-builder` following the guide[Page 203, 1]

To create an agent ramdisk with `Proliant Hardware Manager`, use the `proliant-tools` element in DIB:

```
ironic-python-agent-builder -o proliant-agent-ramdisk -e proliant-tools fedora
```

See the proliant-tools for more information on creating agent ramdisk with `proliant-tools` element in DIB.

Firmware based UEFI iSCSI boot from volume support

With Gen9 (UEFI firmware version 1.40 or higher) and Gen10 HPE Proliant servers, the driver supports firmware based UEFI boot of an iSCSI cinder volume.

This feature requires the node to be configured to boot in `UEFI` boot mode, as well as user image should be `UEFI` bootable image, and `PortFast` needs to be enabled in switch configuration for immediate spanning tree forwarding state so it wouldnt take much time setting the iSCSI target as persistent device.

The driver does not support this functionality when in `bios` boot mode. In case the node is configured with `ilo-pxe` or `ilo-ipxe` as boot interface and the boot mode configured on the bare metal is `bios`, the iscsi boot from volume is performed using iPXE. See *Boot From Volume* for more details.

To use this feature, configure the boot mode of the bare metal to `uefi` and configure the corresponding ironic node using the steps given in *Boot From Volume*. In a cloud environment with nodes configured to boot from `bios` and `uefi` boot modes, the virtual media driver only supports uefi boot mode, and that attempting to use iscsi boot at the same time with a bios volume will result in an error.

BIOS configuration support

The `ilo` and `ilo5` hardware types support `ilo` BIOS interface. The support includes providing manual clean steps *apply_configuration* and *factory_reset* to manage supported BIOS settings on the node. See *BIOS Configuration* for more details and examples.

Note: Prior to the Stein release the user is required to reboot the node manually in order for the settings to take into effect. Starting with the Stein release, iLO drivers reboot the node after running clean steps related to the BIOS configuration. The BIOS settings are cached and the clean step is marked as success only if all the requested settings are applied without any failure. If application of any of the settings fails, the clean step is marked as failed and the settings are not cached.

Configuration

Following are the supported BIOS settings and the corresponding brief description for each of the settings. For a detailed description please refer to HPE Integrated Lights-Out REST API Documentation.

- `AdvancedMemProtection`: Configure additional memory protection with ECC (Error Checking and Correcting). Allowed values are `AdvancedEcc`, `OnlineSpareAdvancedEcc`, `MirroredAdvancedEcc`.
- `AutoPowerOn`: Configure the server to automatically power on when AC power is applied to the system. Allowed values are `AlwaysPowerOn`, `AlwaysPowerOff`, `RestoreLastState`.
- `BootMode`: Select the boot mode of the system. Allowed values are `Uefi`, `LegacyBios`
- `BootOrderPolicy`: Configure how the system attempts to boot devices per the Boot Order when no bootable device is found. Allowed values are `RetryIndefinitely`, `AttemptOnce`, `ResetAfterFailed`.
- `CollabPowerControl`: Enables the Operating System to request processor frequency changes even if the Power Regulator option on the server configured for Dynamic Power Savings Mode. Allowed values are `Enabled`, `Disabled`.
- `DynamicPowerCapping`: Configure when the System ROM executes power calibration during the boot process. Allowed values are `Enabled`, `Disabled`, `Auto`.
- `DynamicPowerResponse`: Enable the System BIOS to control processor performance and power states depending on the processor workload. Allowed values are `Fast`, `Slow`.
- `IntelligentProvisioning`: Enable or disable the Intelligent Provisioning functionality. Allowed values are `Enabled`, `Disabled`.
- `IntelPerfMonitoring`: Exposes certain chipset devices that can be used with the Intel Performance Monitoring Toolkit. Allowed values are `Enabled`, `Disabled`.
- `IntelProcVtd`: Hypervisor or operating system supporting this option can use hardware capabilities provided by Intels Virtualization Technology for Directed I/O. Allowed values are `Enabled`, `Disabled`.
- `IntelQpiFreq`: Set the QPI Link frequency to a lower speed. Allowed values are `Auto`, `MinQpiSpeed`.
- `IntelTxt`: Option to modify Intel TXT support. Allowed values are `Enabled`, `Disabled`.

- `PowerProfile`: Set the power profile to be used. Allowed values are `BalancedPowerPerf`, `MinPower`, `MaxPerf`, `Custom`.
- `PowerRegulator`: Determines how to regulate the power consumption. Allowed values are `DynamicPowerSavings`, `StaticLowPower`, `StaticHighPerf`, `OsControl`.
- `ProcAes`: Enable or disable the Advanced Encryption Standard Instruction Set (AES-NI) in the processor. Allowed values are `Enabled`, `Disabled`.
- `ProcCoreDisable`: Disable processor cores using Intels Core Multi-Processing (CMP) Technology. Allowed values are Integers ranging from `0` to `24`.
- `ProcHyperthreading`: Enable or disable Intel Hyperthreading. Allowed values are `Enabled`, `Disabled`.
- `ProcNoExecute`: Protect your system against malicious code and viruses. Allowed values are `Enabled`, `Disabled`.
- `ProcTurbo`: Enables the processor to transition to a higher frequency than the processors rated speed using Turbo Boost Technology if the processor has available power and is within temperature specifications. Allowed values are `Enabled`, `Disabled`.
- `ProcVirtualization`: Enables or Disables a hypervisor or operating system supporting this option to use hardware capabilities provided by Intels Virtualization Technology. Allowed values are `Enabled`, `Disabled`.
- `SecureBootStatus`: The current state of Secure Boot configuration. Allowed values are `Enabled`, `Disabled`.

Note: This setting is read-only and cant be modified with `apply_configuration` clean step.

- `Sriov`: If enabled, SR-IOV support enables a hypervisor to create virtual instances of a PCI-express device, potentially increasing performance. If enabled, the BIOS allocates additional resources to PCI-express devices. Allowed values are `Enabled`, `Disabled`.
- `ThermalConfig`: select the fan cooling solution for the system. Allowed values are `OptimalCooling`, `IncreasedCooling`, `MaxCooling`
- `ThermalShutdown`: Control the reaction of the system to caution level thermal events. Allowed values are `Enabled`, `Disabled`.
- `TpmState`: Current TPM device state. Allowed values are `NotPresent`, `PresentDisabled`, `PresentEnabled`.

Note: This setting is read-only and cant be modified with `apply_configuration` clean step.

- `TpmType`: Current TPM device type. Allowed values are `NoTpm`, `Tpm12`, `Tpm20`, `Tm10`.

Note: This setting is read-only and cant be modified with `apply_configuration` clean step.

- `UefiOptimizedBoot`: Enables or Disables the System BIOS boot using native UEFI graphics drivers. Allowed values are `Enabled`, `Disabled`.

- `WorkloadProfile`: Change the Workload Profile to accomodate your desired workload. Allowed values are `GeneralPowerEfficientCompute`, `GeneralPeakFrequencyCompute`, `GeneralThroughputCompute`, `Virtualization-PowerEfficient`, `Virtualization-MaxPerformance`, `LowLatency`, `MissionCritical`, `TransactionalApplicationProcessing`, `HighPerformanceCompute`, `DecisionSupport`, `GraphicProcessing`, `I/OThroughput`, `Custom`

Note: This setting is only applicable to ProLiant Gen10 servers with iLO 5 management systems.

Certificate based validation in iLO

The driver supports validation of certificates on the HPE Proliant servers. The path to certificate file needs to be appropriately set in `ca_file` in the nodes `driver_info`. To update SSL certificates into iLO, refer to HPE Integrated Lights-Out Security Technology Brief. Use iLO hostname or IP address as a Common Name (CN) while generating Certificate Signing Request (CSR). Use the same value as *ilo_address* while enrolling node to Bare Metal service to avoid SSL certificate validation errors related to hostname mismatch.

Rescue mode support

The hardware type `ilo` supports rescue functionality. Rescue operation can be used to boot nodes into a rescue ramdisk so that the `rescue` user can access the node.

Please refer to *Rescue Mode* for detailed explanation of rescue feature.

Inject NMI support

The management interface `ilo` supports injection of non-maskable interrupt (NMI) to a bare metal. Following command can be used to inject NMI on a server:

```
baremetal node inject nmi <node>
```

Following command can be used to inject NMI via Compute service:

```
openstack server dump create <server>
```

Note: This feature is supported on HPE ProLiant Gen9 servers and beyond.

Soft power operation support

The power interface `ilo` supports soft power off and soft reboot operations on a bare metal. Following commands can be used to perform soft power operations on a server:

```
baremetal node reboot --soft \
    [--power-timeout <power-timeout>] <node>

baremetal node power off --soft \
    [--power-timeout <power-timeout>] <node>
```

Note: The configuration `[conductor]soft_power_off_timeout` is used as a default timeout value when no timeout is provided while invoking hard or soft power operations.

Note: Server POST state is used to track the power status of HPE ProLiant Gen9 servers and beyond.

Out of Band RAID Support

With Gen10 HPE Proliant servers and later the `ilo5` hardware type supports firmware based RAID configuration as a clean step. This feature requires the node to be configured to `ilo5` hardware type and its raid interface to be `ilo5`. See *RAID Configuration* for more information.

After a successful RAID configuration, the Bare Metal service will update the node with the following information:

- Node `properties/local_gb` is set to the size of root volume.
- Node `properties/root_device` is filled with `wwn` details of root volume. It is used by iLO driver as root device hint during provisioning.

Later the value of raid level of root volume can be added in `baremetal-with-RAID10` (RAID10 for raid level 10) resource class. And consequently flavor needs to be updated to request the resource class to create the server using selected node:

```
baremetal node set test_node --resource-class \
baremetal-with-RAID10

openstack flavor set --property \
resources:CUSTOM_BAREMETAL_WITH_RAID10=1 test-flavor

openstack server create --flavor test-flavor --image test-image instance-1
```

Note: Supported raid levels for `ilo5` hardware type are: 0, 1, 5, 6, 10, 50, 60

IPv6 support

With the IPv6 support in `proliantutils>=2.8.0`, nodes can be enrolled into the baremetal service using the iLO IPv6 addresses.

```
baremetal node create --driver ilo  --deploy-interface direct \
    --driver-info ilo_address=2001:0db8:85a3:0000:0000:8a2e:0370:7334 \
    --driver-info ilo_username=test-user \
    --driver-info ilo_password=test-password \
    --driver-info deploy_iso=test-iso \
    --driver-info rescue_iso=test-iso
```

Note: No configuration changes (in e.g. ironic.conf) are required in order to support IPv6.

Out of Band Sanitize Disk Erase Support

With Gen10 HPE Proliant servers and later the `ilo5` hardware type supports firmware based sanitize disk erase as a clean step. This feature requires the node to be configured to `ilo5` hardware type and its management interface to be `ilo5`.

The possible erase pattern its supports are:

- For HDD - overwrite, zero, crypto
- For SSD - block, zero, crypto

The default erase pattern are, for HDD, overwrite and for SSD, block.

Note: In average 300GB HDD with default pattern overwrite would take approx. 9 hours and 300GB SSD with default pattern block would take approx. 30 seconds to complete the erase.

Out of Band One Button Secure Erase Support

With Gen10 HPE Proliant servers which have been updated with SPP version 2019.03.0 or later the `ilo5` hardware type supports firmware based one button secure erase as a clean step.

The One Button Secure Erase resets iLO and deletes all licenses stored there, resets BIOS settings, and deletes all Active Health System (AHS) and warranty data stored on the system. It also erases supported non-volatile storage data and deletes any deployment settings profiles. See HPE Gen10 Security Reference Guide for more information.

Below are the steps to perform this clean step:

- Perform the cleaning using one_button_secure_erase clean step

```
baremetal node clean $node_ident --clean-steps\
    '[{"interface": "management", "step": "one_button_secure_erase"}]'
```

- Once the clean step would triggered and node go to clean wait state and maintenance flag on node would be set to True, then delete the node

```
baremetal node delete $node_ident
```

Note:

- Even after deleting the node, One Button Secure Erase operation would continue on the node.
- This clean step should be kept last if the multiple clean steps are to be executed. No clean step after this step would be executed.
- One Button Secure Erase should be used with extreme caution, and only when a system is being decommissioned. During the erase the iLO network would keep disconnecting and after the erase user will completely lose iLO access along with the credentials of the server, which needs to be regained by the administrator. The process can take up to a day or two to fully erase and reset all user data.
- When you activate One Button Secure Erase, iLO 5 does not allow firmware update or reset operations.

Note: Do not perform any iLO 5 configuration changes until this process is completed.

UEFI-HTTPS Boot support

The UEFI firmware on Gen10 HPE Proliant servers supports booting from secured URLs. With this capability `ilo5` hardware with `ilo-uefi-https` boot interface supports deploy/rescue features in more secured environments.

If swift is used as glance backend and ironic is configured to use swift to store temporary images, it is required that swift is configured on HTTPS so that the tempurl generated is HTTPS URL.

If the webserver is used for hosting the temporary images, then the webserver is required to serve requests on HTTPS.

If the images are hosted on a HTTPS webserver or swift configured with HTTPS with custom certificates, the user is required to export SSL certificates into iLO. Refer to HPE Integrated Lights-Out Security Technology Brief for more information.

The following command can be used to enroll a ProLiant node with `ilo5` hardware type and `ilo-uefi-https` boot interface:

```
baremetal node create \
    --driver ilo5 \
    --boot-interface ilo-uefi-https \
    --deploy-interface direct \
    --raid-interface ilo5 \
    --rescue-interface agent \
    --driver-info ilo_address=<ilo-ip-address> \
    --driver-info ilo_username=<ilo-username> \
    --driver-info ilo_password=<ilo-password> \
    --driver-info deploy_kernel=<glance-uuid-of-deploy-kernel> \
```

(continues on next page)

(continued from previous page)

```
    --driver-info deploy_ramdisk=<glance-uuid-of-rescue-ramdisk> \
    --driver-info bootloader=<glance-uuid-of-bootloader>
```

Layer 3 or DHCP-less ramdisk booting

DHCP-less deploy is supported by `ilo` and `ilo5` hardware types. However it would work only with ilo-virtual-media boot interface. See *Layer 3 or DHCP-less ramdisk booting* for more information.

Intel IPMI driver

Overview

The `intel-ipmi` hardware type is same as the *IPMI driver* hardware type except for the support of Intel Speed Select Performance Profile (Intel SST-PP) feature. Intel SST-PP allows a server to run different workloads by configuring the CPU to run at 3 distinct operating points or profiles.

Intel SST-PP supports three configuration levels:

- 0 - Intel SST-PP Base Config
- 1 - Intel SST-PP Config 1
- 2 - Intel SST-PP Config 2

The following table shows the list of active cores and their base frequency at different SST-PP config levels:

Config	Cores	Base Freq (GHz)
Base	24	2.4
Config 1	20	2.5
Config 2	16	2.7

This configuration is managed by the management interface `intel-ipmitool` for IntelIPMI hardware.

IntelIPMI manages nodes by using IPMI (Intelligent Platform Management Interface) protocol versions 2.0 or 1.5. It uses the IPMItool utility which is an open-source command-line interface (CLI) for controlling IPMI-enabled devices.

Glossary

- IPMI - Intelligent Platform Management Interface.
- Intel SST-PP - Intel Speed Select Performance Profile.

Enabling the IntelIPMI hardware type

Please see *Configuring IPMI support* for the required dependencies.

1. To enable `intel-ipmi` hardware, add the following configuration to your `ironic.conf`:

```
[DEFAULT]
enabled_hardware_types = intel-ipmi
enabled_management_interfaces = intel-ipmitool
```

2. Restart the Ironic conductor service:

```
sudo service ironic-conductor restart

# Or, for RDO:
sudo systemctl restart openstack-ironic-conductor
```

Registering a node with the IntelIPMI driver

Nodes configured to use the IntelIPMI drivers should have the `driver` field set to `intel-ipmi`.

All the configuration value required for IntelIPMI is the same as the IPMI hardware type except the management interface which is `intel-ipmitool`. Refer *IPMI driver* for details.

The `baremetal node create` command can be used to enroll a node with an IntelIPMI driver. For example:

```
baremetal node create --driver intel-ipmi \
    --driver-info ipmi_address=<address> \
    --driver-info ipmi_username=<username> \
    --driver-info ipmi_password=<password>
```

Features of the `intel-ipmi` hardware type

Intel SST-PP

A node with Intel SST-PP can be configured to use it via `configure_intel_speedselect` deploy step. This deploy accepts:

- `intel_speedselect_config`: Hexadecimal code of Intel SST-PP configuration. Accepted values are 0x00, 0x01, 0x02. These values correspond to *Intel SST-PP Config Base*, *Intel SST-PP Config 1*, *Intel SST-PP Config 2* respectively. The input value must be a string.
- `socket_count`: Number of sockets in the node. The input value must be a positive integer (1 by default).

The deploy step issues an IPMI command with the raw code for each socket in the node to set the requested configuration. A reboot is required to reflect the changes.

Each configuration profile is mapped to traits that Ironic understands. Please note that these names are used for example purpose only. Any name can be used. Only the parameter value should match the deploy step `configure_intel_speedselect`.

- 0 - `CUSTOM_INTEL_SPEED_SELECT_CONFIG_BASE`
- 1 - `CUSTOM_INTEL_SPEED_SELECT_CONFIG_1`
- 2 - `CUSTOM_INTEL_SPEED_SELECT_CONFIG_2`

Now to configure a node with Intel SST-PP while provisioning, create deploy templates for each profiles in Ironic.

```
baremetal deploy template create \
   CUSTOM_INTEL_SPEED_SELECT_CONFIG_BASE \
   --steps '[{"interface": "management", "step": "configure_intel_speedselect
↪", "args": {"intel_speedselect_config": "0x00", "socket_count": 2},
↪"priority": 150}]'

baremetal deploy template create \
    CUSTOM_INTEL_SPEED_SELECT_CONFIG_1 \
    --steps '[{"interface": "management", "step": "configure_intel_speedselect
↪", "args": {"intel_speedselect_config": "0x01", "socket_count": 2},
↪"priority": 150}]'

baremetal deploy template create \
   CUSTOM_INTEL_SPEED_SELECT_CONFIG_2 \
   --steps '[{"interface": "management", "step": "configure_intel_speedselect
↪", "args": {"intel_speedselect_config": "0x02", "socket_count": 2},
↪"priority": 150}]'
```

All Intel SST-PP capable nodes should have these traits associated.

```
baremetal node add trait node-0 \
   CUSTOM_INTEL_SPEED_SELECT_CONFIG_BASE \
   CUSTOM_INTEL_SPEED_SELECT_CONFIG_1 \
   CUSTOM_INTEL_SPEED_SELECT_CONFIG_2
```

To trigger the Intel SST-PP configuration during node provisioning, one of the traits can be added to the flavor.

```
openstack flavor set baremetal --property trait:CUSTOM_INTEL_SPEED_SELECT_
↪CONFIG_1=required
```

Finally create a server with `baremetal` flavor to provision a baremetal node with Intel SST-PP profile *Config 1*.

IPMI driver

Overview

The `ipmi` hardware type manage nodes by using IPMI (Intelligent Platform Management Interface) protocol versions 2.0 or 1.5. It uses the IPMItool utility which is an open-source command-line interface (CLI) for controlling IPMI-enabled devices.

Glossary

- IPMI - Intelligent Platform Management Interface.
- IPMB - Intelligent Platform Management Bus/Bridge.
- BMC - Baseboard Management Controller.
- RMCP - Remote Management Control Protocol.

Enabling the IPMI hardware type

Please see *Configuring IPMI support* for the required dependencies.

1. The `ipmi` hardware type is enabled by default starting with the Ocata release. To enable it explicitly, add the following to your `ironic.conf`:

   ```
   [DEFAULT]
   enabled_hardware_types = ipmi
   enabled_management_interfaces = ipmitool,noop
   enabled_power_interfaces = ipmitool
   ```

   Optionally, enable the *vendor passthru interface* and either or both *console interfaces*:

   ```
   [DEFAULT]
   enabled_hardware_types = ipmi
   enabled_console_interfaces = ipmitool-socat,ipmitool-shellinabox,no-
   ↪console
   enabled_management_interfaces = ipmitool,noop
   enabled_power_interfaces = ipmitool
   enabled_vendor_interfaces = ipmitool,no-vendor
   ```

2. Restart the Ironic conductor service.

Please see *Enabling drivers and hardware types* for more details.

Registering a node with the IPMI driver

Nodes configured to use the IPMItool drivers should have the `driver` field set to `ipmi`.

The following configuration value is required and has to be added to the nodes `driver_info` field:

- `ipmi_address`: The IP address or hostname of the BMC.

Other options may be needed to match the configuration of the BMC, the following options are optional, but in most cases, its considered a good practice to have them set:

- `ipmi_username`: The username to access the BMC; defaults to *NULL* user.
- `ipmi_password`: The password to access the BMC; defaults to *NULL*.
- `ipmi_port`: The remote IPMI RMCP port. By default ipmitool will use the port *623*.

Note: It is highly recommend that you setup a username and password for your BMC.

The `baremetal node create` command can be used to enroll a node with an IPMItool-based driver. For example:

```
baremetal node create --driver ipmi \
    --driver-info ipmi_address=<address> \
    --driver-info ipmi_username=<username> \
    --driver-info ipmi_password=<password>
```

Advanced configuration

When a simple configuration such as providing the `address`, `username` and `password` is not enough, the IPMItool driver contains many other options that can be used to address special usages.

Single/Double bridging functionality

Note: A version of IPMItool higher or equal to 1.8.12 is required to use the bridging functionality.

There are two different bridging functionalities supported by the IPMItool-based drivers: *single* bridge and *dual* bridge.

The following configuration values need to be added to the nodes `driver_info` field so bridging can be used:

- `ipmi_bridging`: The bridging type; default is *no*; other supported values are *single* for single bridge or *dual* for double bridge.
- **`ipmi_local_address`: The local IPMB address for bridged requests.** Required only if `ipmi_bridging` is set to *single* or *dual*. This configuration is optional, if not specified it will be auto discovered by IPMItool.
- `ipmi_target_address`: The destination address for bridged requests. Required only if `ipmi_bridging` is set to *single* or *dual*.
- `ipmi_target_channel`: The destination channel for bridged requests. Required only if `ipmi_bridging` is set to *single* or *dual*.

Double bridge specific options:

- `ipmi_transit_address`: The transit address for bridged requests. Required only if `ipmi_bridging` is set to *dual*.
- `ipmi_transit_channel`: The transit channel for bridged requests. Required only if `ipmi_bridging` is set to *dual*.

The parameter `ipmi_bridging` should specify the type of bridging required: *single* or *dual* to access the bare metal node. If the parameter is not specified, the default value will be set to *no*.

The `baremetal node set` command can be used to set the required bridging information to the Ironic node enrolled with the IPMItool driver. For example:

- Single Bridging:

```
baremetal node set <UUID or name> \
    --driver-info ipmi_local_address=<address> \
    --driver-info ipmi_bridging=single \
    --driver-info ipmi_target_channel=<channel> \
    --driver-info ipmi_target_address=<target address>
```

- Double Bridging:

```
baremetal node set <UUID or name> \
    --driver-info ipmi_local_address=<address> \
    --driver-info ipmi_bridging=dual \
    --driver-info ipmi_transit_channel=<transit channel> \
    --driver-info ipmi_transit_address=<transit address> \
    --driver-info ipmi_target_channel=<target channel> \
    --driver-info ipmi_target_address=<target address>
```

Changing the version of the IPMI protocol

The IPMItool-based drivers works with the versions *2.0* and *1.5* of the IPMI protocol. By default, the version *2.0* is used.

In order to change the IPMI protocol version in the bare metal node, the following option needs to be set to the nodes `driver_info` field:

- `ipmi_protocol_version`: The version of the IPMI protocol; default is *2.0*. Supported values are *1.5* or *2.0*.

The `baremetal node set` command can be used to set the desired protocol version:

```
baremetal node set <UUID or name> --driver-info ipmi_protocol_version=
↪<version>
```

Warning: Version *1.5* of the IPMI protocol does not support encryption. Therefore, it is highly recommended that version 2.0 is used.

Cipher suites

IPMI 2.0 introduces support for encryption and allows setting which cipher suite to use. Traditionally, `ipmitool` was using cipher suite 3 by default, but since SHA1 no longer complies with modern security requirement, recent versions (e.g. the one used in RHEL 8.2) are switching to suite 17.

Normally, the cipher suite to use is negotiated with the BMC using the special command. On some hardware the negotiation yields incorrect results and IPMI commands fail with

```
Error in open session response message : no matching cipher suite
Error: Unable to establish IPMI v2 / RMCP+ session
```

Another possible problem is `ipmitool` commands taking very long (tens of seconds or even minutes) because the BMC does not support cipher suite negotiation. In both cases you can specify the required suite yourself, e.g.

```
baremetal node set <UUID or name> --driver-info ipmi_cipher_suite=3
```

In scenarios where the operator cant specify the `ipmi_cipher_suite` for each node, the configuration parameter `[ipmi]/cipher_suite_versions` can be set to a list of cipher suites that will be used, Ironic will attempt to find a value that can be used from the list provided (from last to first):

```
[ipmi]
cipher_suite_versions = ['1','2','3','6','7','8','11','12']
```

To find the suitable values for this configuration, you can check the field *RMCP+ Cipher Suites* after running an `ipmitool` command, e.g:

```
$ ipmitool -I lanplus -H $HOST -U $USER -v -R 12 -N 5  lan print
# output
Set in Progress         : Set Complete
Auth Type Support       : NONE MD2 MD5 PASSWORD OEM
Auth Type Enable        : Callback : NONE MD2 MD5 PASSWORD OEM
IP Address Source       : Static Address
IP Address              : <IP>
Subnet Mask             : <Subnet>
MAC Address             : <MAC>
RMCP+ Cipher Suites     : 0,1,2,3,6,7,8,11,12
```

Warning: Only the cipher suites 3 and 17 are considered secure by the modern standards. Cipher suite 0 means no security at all.

Using a different privilege level

By default Ironic requests the `ADMINISTRATOR` privilege level of all commands. This is the easiest option, but if its not available for you, you can change it to `CALLBACK`, `OPERATOR` or `USER` this way:

```
baremetal node set <UUID or name> --driver-info ipmi_priv_level=OPERATOR
```

You must ensure that the user can still change power state and boot devices.

Static boot order configuration

See *Static boot order configuration*.

Vendor Differences

While the Intelligent Platform Management Interface (IPMI) interface is based upon a defined standard, the Ironic community is aware of at least one vendor which utilizes a non-standard boot device selector. In essence, this could be something as simple as different interpretation of the standard.

As of October 2020, the known difference is with Supermicro hardware where a selector of `0x24`, signifying a *REMOTE* boot device in the standard, must be used when a boot operation from the local disk subsystem is requested **in UEFI mode**. This is contrary to BIOS mode where the same BMCs expect the selector to be a value of `0x08`.

Because the BMC does not respond with any sort of error, nor do we want to risk BMC connectivity issues by explicitly querying all BMCs what vendor it may be before every operation, the vendor can automatically be recorded in the `properties` field `vendor`. When this is set to a value of `supermicro`, Ironic will navigate the UEFI behavior difference enabling the UEFI to be requested with boot to disk.

Example:

```
baremetal node set <UUID or name> \
    --properties vendor="supermicro"
```

Luckily, Ironic will attempt to perform this detection in power synchronization process, and record this value if not already set.

While similar issues may exist when setting the boot mode and target boot device in other vendors BMCs, we are not aware of them at present. Should you encounter such an issue, please feel free to report this via Storyboard, and be sure to include the `chassis bootparam get 5` output value along with the `mc info` output from your BMC.

Example:

```
ipmitool -I lanplus -H <BMC ADDRESS> -U <Username> -P <Password> \
    mc info
ipmitool -I lanplus -H <BMC ADDRESS> -U <Username> -P <Password> \
    chassis bootparam get 5
```

iRMC driver

Overview

The iRMC driver enables control FUJITSU PRIMERGY via ServerView Common Command Interface (SCCI). Support for FUJITSU PRIMERGY servers consists of the `irmc` hardware type and a few hardware interfaces specific for that hardware type.

Prerequisites

- Install python-scciclient and pysnmp packages:

```
$ pip install "python-scciclient>=0.7.2" pysnmp
```

Hardware Type

The `irmc` hardware type is available for FUJITSU PRIMERGY servers. For information on how to enable the `irmc` hardware type, see *Enabling hardware types*.

Hardware interfaces

The `irmc` hardware type overrides the selection of the following hardware interfaces:

- **bios** Supports `irmc` and `no-bios`. The default is `irmc`.
- **boot** Supports `irmc-virtual-media`, `irmc-pxe`, and `pxe`. The default is `irmc-virtual-media`. The `irmc-virtual-media` boot interface enables the virtual media based deploy with IPA (Ironic Python Agent).

  > **Warning:** We deprecated the `pxe` boot interface when used with `irmc` hardware type. Support for this interface will be removed in the future. Instead, use `irmc-pxe`. `irmc-pxe` boot interface was introduced in Pike.

- **console** Supports `ipmitool-socat`, `ipmitool-shellinabox`, and `no-console`. The default is `ipmitool-socat`.
- **inspect** Supports `irmc`, `inspector`, and `no-inspect`. The default is `irmc`.

  > **Note:** Ironic Inspector needs to be present and configured to use `inspector` as the inspect interface.

- **management** Supports only `irmc`.
- **power** Supports `irmc`, which enables power control via ServerView Common Command Interface (SCCI), by default. Also supports `ipmitool`.
- **raid** Supports `irmc`, `no-raid` and `agent`. The default is `no-raid`.

For other hardware interfaces, `irmc` hardware type supports the Bare Metal reference interfaces. For more details about the hardware interfaces and how to enable the desired ones, see *Enabling hardware interfaces*.

Here is a complete configuration example with most of the supported hardware interfaces enabled for `irmc` hardware type.

```
[DEFAULT]
enabled_hardware_types = irmc
```

(continues on next page)

(continued from previous page)

```
enabled_bios_interfaces = irmc
enabled_boot_interfaces = irmc-virtual-media,irmc-pxe
enabled_console_interfaces = ipmitool-socat,ipmitool-shellinabox,no-console
enabled_deploy_interfaces = direct
enabled_inspect_interfaces = irmc,inspector,no-inspect
enabled_management_interfaces = irmc
enabled_network_interfaces = flat,neutron
enabled_power_interfaces = irmc
enabled_raid_interfaces = no-raid,irmc
enabled_storage_interfaces = noop,cinder
enabled_vendor_interfaces = no-vendor,ipmitool
```

Here is a command example to enroll a node with `irmc` hardware type.

```
baremetal node create \
   --bios-interface irmc \
   --boot-interface irmc-pxe \
   --deploy-interface direct \
   --inspect-interface irmc  \
   --raid-interface irmc
```

Node configuration

Configuration via `driver_info`

- Each node is configured for `irmc` hardware type by setting the following ironic node objects properties:
  - `driver_info/irmc_address` property to be `IP address` or `hostname` of the iRMC.
  - `driver_info/irmc_username` property to be `username` for the iRMC with administrator privileges.
  - `driver_info/irmc_password` property to be `password` for irmc_username.
- If `port` in `[irmc]` section of `/etc/ironic/ironic.conf` or `driver_info/irmc_port` is set to 443, `driver_info/irmc_verify_ca` will take effect:

  `driver_info/irmc_verify_ca` property takes one of 4 value (default value is `True`):
  - `True`: When set to `True`, which certification file iRMC driver uses is determined by `requests` Python module.

    Value of `driver_info/irmc_verify_ca` is passed to `verify` argument of functions defined in `requests` Python module. So which certification will be used is depend on behavior of `requests` module. (maybe certification provided by `certifi` Python module)
  - `False`: When set to `False`, iRMC driver wont verify server certification with certification file during HTTPS connection with iRMC. Just stop to verify server certification, but does HTTPS.

> **Warning:** When set to `False`, user must notice that it can result in vulnerable situation. Stopping verification of server certification during HTTPS connection means it cannot prevent Man-in-the-middle attack. When set to `False`, Ironic user must take enough care around infrastructure environment in terms of security. (e.g. make sure network between Ironic conductor and iRMC is secure)

  - string representing filesystem path to directory which contains certification file: In this case, iRMC driver uses certification file stored at specified directory. Ironic conductor must be able to access that directory. For iRMC to recongnize certification file, Ironic user must run `openssl rehash <path_to_dir>`.
  - string representing filesystem path to certification file: In this case, iRMC driver uses certification file specified. Ironic conductor must have access to that file.

- The following properties are also required if `irmc-virtual-media` boot interface is used:
  - `driver_info/deploy_iso` property to be either deploy iso file name, Glance UUID, or Image Service URL.
  - `instance info/boot_iso` property to be either boot iso file name, Glance UUID, or Image Service URL. This is optional property when `boot_option` is set to `netboot`.

Note: The `deploy_iso` and `boot_iso` properties used to be called `irmc_deploy_iso` and `irmc_boot_iso` accordingly before the Xena release.

- The following properties are also required if `irmc` inspect interface is enabled and SNMPv3 inspection is desired.
  - `driver_info/irmc_snmp_user` property to be the SNMPv3 username. SNMPv3 functionality should be enabled for this user on iRMC server side.
  - `driver_info/irmc_snmp_auth_password` property to be the auth protocol pass phrase. The length of pass phrase should be at least 8 characters.
  - `driver_info/irmc_snmp_priv_password` property to be the privacy protocol pass phrase. The length of pass phrase should be at least 8 characters.

Note: When using SNMPv3, python-scciclient in old version (before 0.12.2) can only interact with iRMC with no authentication protocol setted. This means the passwords and protocol settings of the snmp user in iRMC side should all be blank, otherwise python-scciclient will encounter an communication error. If you are using such old version python-scciclient, the `irmc_snmp_auth_password` and `irmc_snmp_priv_password` properties will be ignored. If you want to set passwords, please update python-scciclient to some newer version (>= 0.12.2).

Configuration via `properties`

- Each node is configured for `irmc` hardware type by setting the following ironic node objects properties:
  - `properties/capabilities` property to be `boot_mode:uefi` if UEFI boot is required, or `boot_mode:bios` if Legacy BIOS is required. If this is not set, `default_boot_mode` at `[default]` section in `ironic.conf` will be used.
  - `properties/capabilities` property to be `secure_boot:true` if UEFI Secure Boot is required. Please refer to *UEFI Secure Boot Support* for more information.

Configuration via `ironic.conf`

- All of the nodes are configured by setting the following configuration options in the `[irmc]` section of `/etc/ironic/ironic.conf`:
  - `port`: Port to be used for iRMC operations; either 80 or 443. The default value is 443. Optional.
  - `auth_method`: Authentication method for iRMC operations; either `basic` or `digest`. The default value is `basic`. Optional.
  - `client_timeout`: Timeout (in seconds) for iRMC operations. The default value is 60. Optional.
  - `sensor_method`: Sensor data retrieval method; either `ipmitool` or `scci`. The default value is `ipmitool`. Optional.
- The following options are required if `irmc-virtual-media` boot interface is enabled:
  - `remote_image_share_root`: Ironic conductor nodes `NFS` or `CIFS` root path. The default value is `/remote_image_share_root`.
  - `remote_image_server`: IP of remote image server.
  - `remote_image_share_type`: Share type of virtual media, either `NFS` or `CIFS`. The default is `CIFS`.
  - `remote_image_share_name`: share name of `remote_image_server`. The default value is `share`.
  - `remote_image_user_name`: User name of `remote_image_server`.
  - `remote_image_user_password`: Password of `remote_image_user_name`.
  - `remote_image_user_domain`: Domain name of `remote_image_user_name`.
- The following options are required if `irmc` inspect interface is enabled:
  - `snmp_version`: SNMP protocol version; either `v1`, `v2c` or `v3`. The default value is `v2c`. Optional.
  - `snmp_port`: SNMP port. The default value is `161`. Optional.
  - `snmp_community`: SNMP community required for versions `v1` and `v2c`. The default value is `public`. Optional.
  - `snmp_security`: SNMP security name required for version `v3`. Optional.

  - `snmp_auth_proto`: The SNMPv3 auth protocol. If using iRMC S4 or S5, the valid value of this option is only `sha`. If using iRMC S6, the valid values are `sha256`, `sha384` and `sha512`. The default value is `sha`. Optional.
  - `snmp_priv_proto`: The SNMPv3 privacy protocol. The valid value and the default value are both `aes`. We will add more supported valid values in the future. Optional.

Note: `snmp_security` will be ignored if `driver_info/irmc_snmp_user` is set. `snmp_auth_proto` and `snmp_priv_proto` will be ignored if the version of python-scciclient is before 0.12.2.

Override `ironic.conf` configuration via `driver_info`

- Each node can be further configured by setting the following ironic node objects properties which override the parameter values in `[irmc]` section of `/etc/ironic/ironic.conf`:
  - `driver_info/irmc_port` property overrides `port`.
  - `driver_info/irmc_auth_method` property overrides `auth_method`.
  - `driver_info/irmc_client_timeout` property overrides `client_timeout`.
  - `driver_info/irmc_sensor_method` property overrides `sensor_method`.
  - `driver_info/irmc_snmp_version` property overrides `snmp_version`.
  - `driver_info/irmc_snmp_port` property overrides `snmp_port`.
  - `driver_info/irmc_snmp_community` property overrides `snmp_community`.
  - `driver_info/irmc_snmp_security` property overrides `snmp_security`.
  - `driver_info/irmc_snmp_auth_proto` property overrides `snmp_auth_proto`.
  - `driver_info/irmc_snmp_priv_proto` property overrides `snmp_priv_proto`.

Optional functionalities for the `irmc` hardware type

UEFI Secure Boot Support

The hardware type `irmc` supports secure boot deploy, see *UEFI secure boot mode* for details.

Warning: Secure boot feature is not supported with `pxe` boot interface.

Node Cleaning Support

The `irmc` hardware type supports node cleaning. For more information on node cleaning, see *Node cleaning*.

Supported Automated Cleaning Operations

The automated cleaning operations supported are:

- `restore_irmc_bios_config`: Restores BIOS settings on a baremetal node from backup data. If this clean step is enabled, the BIOS settings of a baremetal node will be backed up automatically before the deployment. By default, this clean step is disabled with priority `0`. Set its priority to a positive integer to enable it. The recommended value is `10`.

  > **Warning:** `pxe` boot interface, when used with `irmc` hardware type, does not support this clean step. If uses `irmc` hardware type, it is required to select `irmc-pxe` or `irmc-virtual-media` as the boot interface in order to make this clean step work.

Configuration options for the automated cleaning steps are listed under `[irmc]` section in ironic.conf

```
clean_priority_restore_irmc_bios_config = 0
```

For more information on node automated cleaning, see *Automated cleaning*

Boot from Remote Volume

The `irmc` hardware type supports the generic PXE-based remote volume booting when using the following boot interfaces:

- `irmc-pxe`
- `pxe`

In addition, the `irmc` hardware type supports remote volume booting without PXE. This is available when using the `irmc-virtual-media` boot interface. This feature configures a node to boot from a remote volume by using the API of iRMC. It supports iSCSI and FibreChannel.

Configuration

In addition to the configuration for generic drivers to *remote volume boot*, the iRMC driver requires the following configuration:

- It is necessary to set physical port IDs to network ports and volume connectors. All cards including those not used for volume boot should be registered.

  The format of a physical port ID is: `<Card Type><Slot No>-<Port No>` where:

  - `<Card Type>`: could be `LAN`, `FC` or `CNA`
  - `<Slot No>`: 0 indicates onboard slot. Use 1 to 9 for add-on slots.
  - `<Port No>`: A port number starting from 1.

These IDs are specified in a nodes `driver_info[irmc_pci_physical_ids]`. This value is a dictionary. The key is the UUID of a resource (Port or Volume Connector) and its value is the physical port ID. For example:

```
{
  "1ecd14ee-c191-4007-8413-16bb5d5a73a2":"LAN0-1",
  "87f6c778-e60e-4df2-bdad-2605d53e6fc0":"CNA1-1"
}
```

It can be set with the following command:

```
baremetal node set $NODE_UUID \
--driver-info irmc_pci_physical_ids={} \
--driver-info irmc_pci_physical_ids/$PORT_UUID=LAN0-1 \
--driver-info irmc_pci_physical_ids/$VOLUME_CONNECTOR_UUID=CNA1-1
```

- For iSCSI boot, volume connectors with both types `iqn` and `ip` are required. The configuration with DHCP is not supported yet.
- For iSCSI, the size of the storage network is needed. This value should be specified in a nodes `driver_info[irmc_storage_network_size]`. It must be a positive integer < 32. For example, if the storage network is 10.2.0.0/22, use the following command:

```
baremetal node set $NODE_UUID --driver-info irmc_storage_network_size=22
```

Supported hardware

The driver supports the PCI controllers, Fibrechannel Cards, Converged Network Adapters supported by Fujitsu ServerView Virtual-IO Manager.

Hardware Inspection Support

The `irmc` hardware type provides the iRMC-specific hardware inspection with `irmc` inspect interface.

Note: SNMP requires being enabled in ServerViewő iRMC S4 Web Server(Network SettingsSNMP section).

Configuration

The Hardware Inspection Support in the iRMC driver requires the following configuration:

- It is necessary to set ironic configuration with `gpu_ids` and `fpga_ids` options in `[irmc]` section.

  `gpu_ids` and `fpga_ids` are lists of `<vendorID>/<deviceID>` where:

  - `<vendorID>`: 4 hexadecimal digits starts with 0x.
  - `<deviceID>`: 4 hexadecimal digits starts with 0x.

  Here are sample values for `gpu_ids` and `fpga_ids`:

```
gpu_ids = 0x1000/0x0079,0x2100/0x0080
fpga_ids = 0x1000/0x005b,0x1100/0x0180
```

- The python-scciclient package requires pyghmi version >= 1.0.22 and pysnmp version >= 4.2.3. They are used by the conductor service on the conductor. The latest version of pyghmi can be downloaded from here and pysnmp can be downloaded from here.

Supported properties

The inspection process will discover the following essential properties (properties required for scheduling deployment):

- `memory_mb`: memory size
- `cpus`: number of cpus
- `cpu_arch`: cpu architecture
- `local_gb`: disk size

Inspection can also discover the following extra capabilities for iRMC driver:

- `irmc_firmware_version`: iRMC firmware version
- `rom_firmware_version`: ROM firmware version
- `trusted_boot`: The flag whether TPM(Trusted Platform Module) is supported by the server. The possible values are True or False.
- `server_model`: server model
- `pci_gpu_devices`: number of gpu devices connected to the bare metal.

Inspection can also set/unset nodes traits with the following cpu type for iRMC driver:

- `CUSTOM_CPU_FPGA`: The bare metal contains fpga cpu type.

Note:

- The disk size is returned only when eLCM License for FUJITSU PRIMERGY servers is activated. If the license is not activated, then Hardware Inspection will fail to get this value.
- Before inspecting, if the server is power-off, it will be turned on automatically. System will wait for a few second before start inspecting. After inspection, power status will be restored to the previous state.

The operator can specify these capabilities in compute service flavor, for example:

```
openstack flavor set baremetal-flavor-name --property capabilities:irmc_
↪firmware_version="iRMC S4-8.64F"

openstack flavor set baremetal-flavor-name --property capabilities:server_
↪model="TX2540M1F5"

openstack flavor set baremetal-flavor-name --property capabilities:pci_gpu_
↪devices="1"
```

See *Capabilities discovery* for more details and examples.

The operator can add a trait in compute service flavor, for example:

```
baremetal node add trait $NODE_UUID CUSTOM_CPU_FPGA
```

A valid trait must be no longer than 255 characters. Standard traits are defined in the os_traits library. A custom trait must start with the prefix `CUSTOM_` and use the following characters: A-Z, 0-9 and _.

RAID configuration Support

The `irmc` hardware type provides the iRMC RAID configuration with `irmc` raid interface.

Note:

- RAID implementation for `irmc` hardware type is based on eLCM license and SDCard. Otherwise, SP(Service Platform) in lifecycle management must be available.
- RAID implementation only supported for RAIDAdapter 0 in Fujitsu Servers.

Configuration

The RAID configuration Support in the iRMC drivers requires the following configuration:

- It is necessary to set ironic configuration into Node with JSON file option:

  ```
  $ baremetal node set <node-uuid-or-name> \
    --target-raid-config <JSON file containing target RAID configuration>
  ```

  Here is some sample values for JSON file:

  ```
  {
      "logical_disks": [
          {
              "size_gb": 1000,
              "raid_level": "1"
      ]
  }
  ```

  or:

  ```
  {
      "logical_disks": [
          {
              "size_gb": 1000,
              "raid_level": "1",
              "controller": "FTS RAID Ctrl SAS 6G 1GB (D3116C) (0)",
              "physical_disks": [
                  "0",
                  "1"
  ```

(continues on next page)

(continued from previous page)

```
            ]
        }
    ]
}
```

Note: RAID 1+0 and 5+0 in iRMC driver does not support property `physical_disks` in `target_raid_config` during create raid configuration yet. See following example:

```
{
  "logical_disks":
    [
      {
        "size_gb": "MAX",
        "raid_level": "1+0"
      }
    ]
}
```

See *RAID Configuration* for more details and examples.

Supported properties

The RAID configuration using iRMC driver supports following parameters in JSON file:

- `size_gb`: is mandatory properties in Ironic.
- `raid_level`: is mandatory properties in Ironic. Currently, iRMC Server supports following RAID levels: 0, 1, 5, 6, 1+0 and 5+0.
- `controller`: is name of the controller as read by the RAID interface.
- `physical_disks`: are specific values for each raid array in LogicalDrive which operator want to set them along with `raid_level`.

The RAID configuration is supported as a manual cleaning step.

Note:

- iRMC server will power-on after create/delete raid configuration is applied, FGI (Foreground Initialize) will process raid configuration in iRMC server, thus the operation will completed upon power-on and power-off when created RAID on iRMC server.

See *RAID Configuration* for more details and examples.

BIOS configuration Support

The `irmc` hardware type provides the iRMC BIOS configuration with `irmc` bios interface.

Warning: `irmc` bios interface does not support `factory_reset`.

Starting from version `0.10.0` of `python-scciclient`, the BIOS setting obtained may not be the latest. If you want to get the latest BIOS setting, you need to delete the existing BIOS profile in iRMC. For example:

```
curl -u user:pass -H "Content-type: application/json" -X DELETE -i http://
↪192.168.0.1/rest/v1/Oem/eLCM/ProfileManagement/BiosConfig
```

Configuration

The BIOS configuration in the iRMC driver supports the following settings:

- `boot_option_filter`: Specifies from which drives can be booted. This supports following options: `UefiAndLegacy`, `LegacyOnly`, `UefiOnly`.
- `check_controllers_health_status_enabled`: The UEFI FW checks the controller health status. This supports following options: `true`, `false`.
- `cpu_active_processor_cores`: The number of active processor cores 1n. Option 0 indicates that all available processor cores are active.
- `cpu_adjacent_cache_line_prefetch_enabled`: The processor loads the requested cache line and the adjacent cache line. This supports following options: `true`, `false`.
- `cpu_vt_enabled`: Supports the virtualization of platform hardware and several software environments, based on Virtual Machine Extensions to support the use of several software environments using virtual computers. This supports following options: `true`, `false`.
- `flash_write_enabled`: The system BIOS can be written. Flash BIOS update is possible. This supports following options: `true`, `false`.
- `hyper_threading_enabled`: Hyper-threading technology allows a single physical processor core to appear as several logical processors. This supports following options: `true`, `false`.
- `keep_void_boot_options_enabled`: Boot Options will not be removed from Boot Option Priority list. This supports following options: `true`, `false`.
- `launch_csm_enabled`: Specifies whether the Compatibility Support Module (CSM) is executed. This supports following options: `true`, `false`.
- `os_energy_performance_override_enabled`: Prevents the OS from overruling any energy efficiency policy setting of the setup. This supports following options: `true`, `false`.
- `pci_aspm_support`: Active State Power Management (ASPM) is used to power-manage the PCI Express links, thus consuming less power. This supports following options: `Disabled`, `Auto`, `L0Limited`, `L1only`, `L0Force`.
- `pci_above_4g_decoding_enabled`: Specifies if memory resources above the 4GB address boundary can be assigned to PCI devices. This supports following options: `true`, `false`.

- `power_on_source`: Specifies whether the switch on sources for the system are managed by the BIOS or the ACPI operating system. This supports following options: `BiosControlled`, `AcpiControlled`.
- `single_root_io_virtualization_support_enabled`: Single Root IO Virtualization Support is active. This supports following options: `true`, `false`.

The BIOS configuration is supported as a manual cleaning step. See *BIOS Configuration* for more details and examples.

Supported platforms

This driver supports FUJITSU PRIMERGY RX M4 servers and above.

When `irmc` power interface is used, Soft Reboot (Graceful Reset) and Soft Power Off (Graceful Power Off) are only available if ServerView agents are installed. See iRMC S4 Manual for more details.

RAID configuration feature supports FUJITSU PRIMERGY servers with RAID-Ctrl-SAS-6G-1GB(D3116C) controller and above. For detail supported controller with OOB-RAID configuration, please see the whitepaper for iRMC RAID configuration.

Redfish driver

Overview

The `redfish` driver enables managing servers compliant with the Redfish protocol.

Prerequisites

- The Sushy library should be installed on the ironic conductor node(s).

  For example, it can be installed with `pip`:

```
sudo pip install sushy
```

Enabling the Redfish driver

1. Add `redfish` to the list of `enabled_hardware_types`, `enabled_power_interfaces`, `enabled_management_interfaces` and `enabled_inspect_interfaces` as well as `redfish-virtual-media` to `enabled_boot_interfaces` in `/etc/ironic/ironic.conf`. For example:

```
[DEFAULT]
...
enabled_hardware_types = ipmi,redfish
enabled_boot_interfaces = ipxe,redfish-virtual-media
enabled_power_interfaces = ipmitool,redfish
enabled_management_interfaces = ipmitool,redfish
enabled_inspect_interfaces = inspector,redfish
```

2. Restart the ironic conductor service:

```
sudo service ironic-conductor restart

# Or, for RDO:
sudo systemctl restart openstack-ironic-conductor
```

Registering a node with the Redfish driver

Nodes configured to use the driver should have the `driver` property set to `redfish`.

The following properties are specified in the nodes `driver_info` field:

- **`redfish_address`: The URL address to the Redfish controller. It must** include the authority portion of the URL, and can optionally include the scheme. If the scheme is missing, https is assumed. For example: https://mgmt.vendor.com. This is required.
- **`redfish_system_id`: The canonical path to the ComputerSystem resource** that the driver will interact with. It should include the root service, version and the unique resource path to the ComputerSystem. This property is only required if target BMC manages more than one ComputerSystem. Otherwise ironic will pick the only available ComputerSystem automatically. For example: /redfish/v1/Systems/1.
- **`redfish_username`: User account with admin/server-profile access** privilege. Although not required, it is highly recommended.
- **`redfish_password`: User account password. Although not required, it is** highly recommended.
- **`redfish_verify_ca`: If redfish_address has the https scheme, the** driver will use a secure (TLS) connection when talking to the Redfish controller. By default (if this is not set or set to True), the driver will try to verify the host certificates. This can be set to the path of a certificate file or directory with trusted certificates that the driver will use for verification. To disable verifying TLS, set this to False. This is optional.
- **`redfish_auth_type`: Redfish HTTP client authentication method. Can be** basic, session or auto. The auto mode first tries session and falls back to basic if session authentication is not supported by the Redfish BMC. Default is set in ironic config as `[redfish]auth_type`. Most operators should not need to leverage this setting. Session based authentication should generally be used in most cases as it prevents re-authentication every time a background task checks in with the BMC.

Note: The `redfish_address`, `redfish_username`, `redfish_password`, and `redfish_verify_ca` fields, if changed, will trigger a new session to be establsihed and cached with the BMC. The `redfish_auth_type` field will only be used for the creation of a new cached session, or should one be rejected by the BMC.

The `baremetal node create` command can be used to enroll a node with the `redfish` driver. For example:

```
baremetal node create --driver redfish --driver-info \
  redfish_address=https://example.com --driver-info \
```

(continues on next page)

(continued from previous page)

```
  redfish_system_id=/redfish/v1/Systems/CX34R87 --driver-info \
  redfish_username=admin --driver-info redfish_password=password \
  --name node-0
```

For more information about enrolling nodes see *Enrollment* in the install guide.

Boot mode support

The `redfish` hardware type can read current boot mode from the bare metal node as well as set it to either Legacy BIOS or UEFI.

Note: Boot mode management is the optional part of the Redfish specification. Not all Redfish-compliant BMCs might implement it. In that case it remains the responsibility of the operator to configure proper boot mode to their bare metal nodes.

UEFI secure boot

Secure boot mode can be automatically set and unset during deployment for nodes in UEFI boot mode, see *UEFI secure boot mode* for an explanation how to use it.

Two clean and deploy steps are provided for key management:

`management.reset_secure_boot_keys_to_default` resets secure boot keys to their manufacturing defaults.

`management.clear_secure_boot_keys` removes all secure boot keys from the node.

Out-Of-Band inspection

The `redfish` hardware type can inspect the bare metal node by querying Redfish compatible BMC. This process is quick and reliable compared to the way the `inspector` hardware type works i.e. booting bare metal node into the introspection ramdisk.

Note: The `redfish` inspect interface relies on the optional parts of the Redfish specification. Not all Redfish-compliant BMCs might serve the required information, in which case bare metal node inspection will fail.

Note: The `local_gb` property cannot always be discovered, for example, when a node does not have local storage or the Redfish implementation does not support the required schema. In this case the property will be set to 0.

Virtual media boot

The idea behind virtual media boot is that BMC gets hold of the boot image one way or the other (e.g. by HTTP GET, other methods are defined in the standard), then inserts it into nodes virtual drive as if it was burnt on a physical CD/DVD. The node can then boot from that virtual drive into the operating system residing on the image.

The major advantage of virtual media boot feature is that potentially unreliable TFTP image transfer phase of PXE protocol suite is fully eliminated.

Hardware types based on the `redfish` fully support booting deploy/rescue and user images over virtual media. Ironic builds bootable ISO images, for either UEFI or BIOS (Legacy) boot modes, at the moment of node deployment out of kernel and ramdisk images associated with the ironic node.

To boot a node managed by `redfish` hardware type over virtual media using BIOS boot mode, it suffice to set ironic boot interface to `redfish-virtual-media`, as opposed to `ipmitool`.

```
baremetal node set --boot-interface redfish-virtual-media node-0
```

Note: iDRAC firmware before 4.40.10.00 (on Intel systems) and 6.00.00.00 (on AMD systems) requires a non-standard Redfish call to boot from virtual media. Consider upgrading to 6.00.00.00, otherwise you **must** use the `idrac` hardware type and the `idrac-redfish-virtual-media` boot interface with older iDRAC firmware instead. For simplicity Ironic restricts both AMD and Intel systems before firmware version 6.00.00.00. See *iDRAC driver* for more details on this hardware type.

If UEFI boot mode is desired, the user should additionally supply EFI System Partition image (ESP), see *Configuring an ESP image* for details.

If `[driver_info]/config_via_floppy` boolean property of the node is set to `true`, ironic will create a file with runtime configuration parameters, place into on a FAT image, then insert the image into nodes virtual floppy drive.

When booting over PXE or virtual media, and user instance requires some specific kernel configuration, the nodes `instance_info[kernel_append_params]` or `driver_info[kernel_append_params]` properties can be used to pass user-specified kernel command line parameters.

```
baremetal node set node-0 \
  --driver-info kernel_append_params="nofb nomodeset vga=normal"
```

Note: The `driver_info` field is supported starting with the Xena release.

For ramdisk boot, the `instance_info[ramdisk_kernel_arguments]` property serves the same purpose.

Pre-built ISO images

By default an ISO images is built per node using the deploy kernel and initramfs provided in the configuration or the nodes `driver_info`. Starting with the Wallaby release its possible to provide a pre-built ISO image:

```
baremetal node set node-0 \
  --driver_info deploy_iso=http://url/of/deploy.iso \
  --driver_info rescue_iso=http://url/of/rescue.iso
```

Note: OpenStack Image service (glance) image IDs and `file://` links are also accepted.

Note: Before the Xena release the parameters were called `redfish_deploy_iso` and `redfish_rescue_iso` accordingly. The old names are still supported for backward compatibility.

No customization is currently done to the image, so e.g. *Layer 3 or DHCP-less ramdisk booting* wont work. *Configuring an ESP image* is also unnecessary.

Configuring an ESP image

An ESP image is an image that contains the necessary bootloader to boot the ISO in UEFI mode. You will need a GRUB2 image file, as well as Shim for secure boot. See *UEFI PXE - Grub setup* for an explanation how to get them.

Then the following script can be used to build an ESP image:

```
DEST=/path/to/esp.img
GRUB2=/path/to/grub.efi
SHIM=/path/to/shim.efi
TEMP_MOUNT=$(mktemp -d)

dd if=/dev/zero of=$DEST bs=4096 count=1024
mkfs.fat -s 4 -r 512 -S 4096 $DEST

sudo mount $DEST $TEMP_MOUNT
sudo mkdir -p $DEST/EFI/BOOT
sudo cp "$SHIM" $DEST/EFI/BOOT/BOOTX64.efi
sudo cp "$GRUB2" $DEST/EFI/BOOT/GRUBX64.efi
sudo umount $TEMP_MOUNT
```

Note: If you use an architecture other than x86-64, youll need to adjust the destination paths.

The resulting image should be provided via the `driver_info/bootloader` ironic node property in form of an image UUID or a URL:

```
baremetal node set --driver-info bootloader=<glance-uuid-or-url> node-0
```

Alternatively, set the bootloader UUID or URL in the configuration file:

```
[conductor]
bootloader = <glance-uuid-or-url>
```

Finally, you need to provide the correct GRUB2 configuration path for your image. In most cases this path will depend on your distribution, more precisely, the distribution you took the GRUB2 image from. For example:

CentOS:

```
[DEFAULT]
grub_config_path = EFI/centos/grub.cfg
```

Ubuntu:

```
[DEFAULT]
grub_config_path = EFI/ubuntu/grub.cfg
```

Note: Unlike in the script above, these paths are case-sensitive!

Virtual Media Ramdisk

The `ramdisk` deploy interface can be used in concert with the `redfish-virtual-media` boot interface to facilitate the boot of a remote node utilizing pre-supplied virtual media. See *Booting a Ramdisk or an ISO* for information on how to enable and configure it.

Instead of supplying an `[instance_info]/image_source` parameter, a `[instance_info]/boot_iso` parameter can be supplied. The image will be downloaded by the conductor, and the instance will be booted using the supplied ISO image. In accordance with the `ramdisk` deployment interface behavior, once booted the machine will have a `provision_state` of `ACTIVE`.

```
baremetal node set <node name or UUID> \
    --boot-interface redfish-virtual-media \
    --deploy-interface ramdisk \
    --instance_info boot_iso=http://url/to.iso
```

This initial interface does not support bootloader configuration parameter injection, as such the `[instance_info]/kernel_append_params` setting is ignored.

Configuration drives are supported starting with the Wallaby release for nodes that have a free virtual USB slot:

```
baremetal node deploy <node name or UUID> \
    --config-drive '{"meta_data": {...}, "user_data": "..."}'
```

or via a link to a raw image:

```
baremetal node deploy <node name or UUID> \
    --config-drive http://example.com/config.img
```

Layer 3 or DHCP-less ramdisk booting

DHCP-less deploy is supported by the Redfish virtual media boot. See *Layer 3 or DHCP-less ramdisk booting* for more information.

Firmware update using manual cleaning

The `redfish` hardware type supports updating the firmware on nodes using a manual cleaning step.

The firmware update cleaning step allows one or more firmware updates to be applied to a node. If multiple updates are specified, then they are applied sequentially in the order given. The server is rebooted once per update. If a failure occurs, the cleaning step immediately fails which may result in some updates not being applied. If the node is placed into maintenance mode while a firmware update cleaning step is running that is performing multiple firmware updates, the update in progress will complete, and processing of the remaining updates will pause. When the node is taken out of maintenance mode, processing of the remaining updates will continue.

When updating the BMC firmware, the BMC may become unavailable for a period of time as it resets. In this case, it may be desireable to have the cleaning step wait after the update has been applied before indicating that the update was successful. This allows the BMC time to fully reset before further operations are carried out against it. To cause the cleaning step to wait after applying an update, an optional `wait` argument may be specified in the firmware image dictionary. The value of this argument indicates the number of seconds to wait following the update. If the `wait` argument is not specified, then this is equivalent to `wait 0`, meaning that it will not wait and immediately proceed with the next firmware update if there is one, or complete the cleaning step if not.

The `update_firmware` cleaning step accepts JSON in the following format:

```
[{
    "interface": "management",
    "step": "update_firmware",
    "args": {
        "firmware_images":[
            {
                "url": "<url_to_firmware_image1>",
                "checksum": "<checksum for image, uses SHA1>",
                "source": "<optional override source setting for image>",
                "wait": <number_of_seconds_to_wait>
            },
            {
                "url": "<url_to_firmware_image2>"
            },
            ...
        ]
    }
}]
```

The different attributes of the `update_firmware` cleaning step are as follows:

Attribute	Description
`interface`	Interface of the cleaning step. Must be `management` for firmware update
`step`	Name of cleaning step. Must be `update_firmware` for firmware update
`args`	Keyword-argument entry (<name>: <value>) being passed to cleaning step
`args.firmware_images`	Ordered list of dictionaries of firmware images to be applied

Each firmware image dictionary, is of the form:

```
{
  "url": "<URL of firmware image file>",
  "checksum": "<checksum for image, uses SHA1>",
  "source": "<Optional override source setting for image>",
  "wait": <Optional time in seconds to wait after applying update>
}
```

The `url` and `checksum` arguments in the firmware image dictionary are mandatory, while the `source` and `wait` arguments are optional.

For `url` currently `http`, `https`, `swift` and `file` schemes are supported.

`source` corresponds to `[redfish]firmware_source` and by setting it here, it is possible to override global setting per firmware image in clean step arguments.

Note: At the present time, targets for the firmware update cannot be specified. In testing, the BMC applied the update to all applicable targets on the node. It is assumed that the BMC knows what components a given firmware image is applicable to.

To perform a firmware update, first download the firmware to a web server, Swift or filesystem that the Ironic conductor or BMC has network access to. This could be the ironic conductor web server or another web server on the BMC network. Using a web browser, curl, or similar tool on a server that has network access to the BMC or Ironic conductor, try downloading the firmware to verify that the URLs are correct and that the web server is configured properly.

Next, construct the JSON for the firmware update cleaning step to be executed. When launching the firmware update, the JSON may be specified on the command line directly or in a file. The following example shows one cleaning step that installs four firmware updates. All except 3rd entry that has explicit `source` added, uses setting from `[redfish]firmware_source` to determine if and where to stage the files:

```
[{
    "interface": "management",
    "step": "update_firmware",
    "args": {
        "firmware_images":[
            {
                "url": "http://192.0.2.10/BMC_4_22_00_00.EXE",
                "checksum": "<sha1-checksum-of-the-file>",
                "wait": 300
            },
```

(continues on next page)

(continued from previous page)

```
            {
                "url": "https://192.0.2.10/NIC_19.0.12_A00.EXE",
                "checksum": "<sha1-checksum-of-the-file>"
            },
            {
                "url": "file:///firmware_images/idrac/9/PERC_WN64_6.65.65.65_
↪A00.EXE",
                "checksum": "<sha1-checksum-of-the-file>",
                "source": "http"
            },
            {
                "url": "swift://firmware_container/BIOS_W8Y0W_WN64_2.1.7.EXE",
                "checksum": "<sha1-checksum-of-the-file>"
            }
        ]
    }
}]
```

Finally, launch the firmware update cleaning step against the node. The following example assumes the above JSON is in a file named `firmware_update.json`:

```
baremetal node clean <ironic_node_uuid> --clean-steps firmware_update.json
```

In the following example, the JSON is specified directly on the command line:

```
baremetal node clean <ironic_node_uuid> --clean-steps '[{"interface":
↪"management", "step": "update_firmware", "args": {"firmware_images":[{"url
↪": "http://192.0.2.10/BMC_4_22_00_00.EXE", "wait": 300}, {"url": "https://
↪192.0.2.10/NIC_19.0.12_A00.EXE"}]}}]'
```

Note: Firmware updates may take some time to complete. If a firmware update cleaning step consistently times out, then consider performing fewer firmware updates in the cleaning step or increasing `clean_callback_timeout` in ironic.conf to increase the timeout value.

Warning: Warning: Removing power from a server while it is in the process of updating firmware may result in devices in the server, or the server itself becoming inoperable.

Retrieving BIOS Settings

When the *bios interface* is set to `redfish`, Ironic will retrieve the nodes BIOS settings as described in BIOS Configuration. In addition, via Sushy, Ironic will get the BIOS Attribute Registry (BIOS Registry) from the node which is a schema providing details on the settings. The following fields will be returned in the BIOS API (`/v1/nodes/{node_ident}/bios`) along with the setting name and value:

Field	Description
`attribute_type`	The type of setting - `Enumeration`, `Integer`, `String`, `Boolean`, or `Password`
`allowable_values`	A list of allowable values when the attribute_type is `Enumeration`
`lower_bound`	The lowest allowed value when attribute_type is `Integer`
`upper_bound`	The highest allowed value when attribute_type is `Integer`
`min_length`	The shortest string length that the value can have when attribute_type is `String`
`max_length`	The longest string length that the value can have when attribute_type is `String`
`read_only`	The setting is ready only and cannot be modified
`unique`	The setting is specific to this node
`reset_required`	After changing this setting a node reboot is required

Node Vendor Passthru Methods

Method	Description
`create_subscription`	Create a new subscription on the Node
`delete_subscription`	Delete a subscription of a Node
`get_all_subscriptions`	List all subscriptions of a Node
`get_subscription`	Show a single subscription of a Node
`eject_vmedia`	Eject attached virtual media from a Node

Create Subscription

Table 1: Request

Name	In	Type	Description
Destination	body	string	The URI of the destination Event Service
EventTypes (optional)	body	array	List of ypes of events that shall be sent to the destination
Context (optional)	body	string	A client-supplied string that is stored with the event destination subscription
Protocol (optional)	body	string	The protocol type that the event will use for sending the event to the destination

Example JSON to use in `create_subscription`:

```
{
    "Destination": "https://someurl",
    "EventTypes": ["Alert"],
    "Context": "MyProtocol",
    "args": "Redfish"
}
```

Delete Subscription

Table 2: Request

Name	In	Type	Description
id	body	string	The Id of the subscription generated by the BMC

Example JSON to use in `delete_subscription`:

```
{
    "id": "<id of the subscription generated by the BMC>"
}
```

Get Subscription

Table 3: Request

Name	In	Type	Description
id	body	string	The Id of the subscription generated by the BMC

Example JSON to use in `get_subscription`:

```
{
    "id": "<id of the subscription generated by the BMC>"
}
```

Get All Subscriptions

The `get_all_subscriptions` doesnt require any parameters.

Eject Virtual Media

Table 4: Request

Name	In	Type	Description
boot_device (optional)	body	string	Type of the device to eject (all devices by default)

Internal Session Cache

The `redfish` hardware type, and derived interfaces, utilizes a built-in session cache which prevents Ironic from re-authenticating every time Ironic attempts to connect to the BMC for any reason.

This consists of cached connectors objects which are used and tracked by a unique consideration of `redfish_username`, `redfish_password`, `redfish_verify_ca`, and finally `redfish_address`. Changing any one of those values will trigger a new session to be created. The `redfish_system_id` value is explicitly not considered as Redfish has a model of use of one BMC to many systems, which is also a model Ironic supports.

The session cache default size is `1000` sessions per conductor. If you are operating a deployment with a larger number of Redfish BMCs, it is advised that you do appropriately tune that number. This can be tuned via the API service configuration file, `[redfish]connection_cache_size`.

Session Cache Expiration

By default, sessions remain cached for as long as possible in memory, as long as they have not experienced an authentication, connection, or other unexplained error.

Under normal circumstances, the sessions will only be rolled out of the cache in order of oldest first when the cache becomes full. There is no time based expiration to entries in the session cache.

Of course, the cache is only in memory, and restarting the `ironic-conductor` will also cause the cache to be rebuilt from scratch. If this is due to any persistent connectivity issue, this may be sign of an unexpected condition, and please consider contacting the Ironic developer community for assistance.

SNMP driver

The SNMP hardware type enables control of power distribution units of the type frequently found in data centre racks. PDUs frequently have a management ethernet interface and SNMP support enabling control of the power outlets.

The SNMP power interface works with the *PXE boot* interface for network deployment and network-configured boot.

Note: Unlike most of the other power interfaces, the SNMP power interface does not have a corresponding management interface. The SNMP hardware type uses the `noop` management interface instead.

List of supported devices

This is a non-exhaustive list of supported devices. Any device not listed in this table could possibly work using a similar driver.

Please report any device status.

Manufacturer	Model	Supported?	Driver name
APC	AP7920	Yes	apc_masterswitch
APC	AP9606	Yes	apc_masterswitch
APC	AP9225	Yes	apc_masterswitchplus
APC	AP7155	Yes	apc_rackpdu
APC	AP7900	Yes	apc_rackpdu
APC	AP7901	Yes	apc_rackpdu
APC	AP7902	Yes	apc_rackpdu
APC	AP7911a	Yes	apc_rackpdu
APC	AP7921	Yes	apc_rackpdu
APC	AP7922	Yes	apc_rackpdu
APC	AP7930	Yes	apc_rackpdu
APC	AP7931	Yes	apc_rackpdu
APC	AP7932	Yes	apc_rackpdu
APC	AP7940	Yes	apc_rackpdu
APC	AP7941	Yes	apc_rackpdu
APC	AP7951	Yes	apc_rackpdu
APC	AP7960	Yes	apc_rackpdu
APC	AP7990	Yes	apc_rackpdu
APC	AP7998	Yes	apc_rackpdu
APC	AP8941	Yes	apc_rackpdu
APC	AP8953	Yes	apc_rackpdu
APC	AP8959	Yes	apc_rackpdu
APC	AP8961	Yes	apc_rackpdu
APC	AP8965	Yes	apc_rackpdu
Aten	all?	Yes	aten
CyberPower	all?	Untested	cyberpower
EatonPower	all?	Untested	eatonpower
Teltronix	all?	Yes	teltronix
BayTech	MRP27	Yes	baytech_mrp27

Software Requirements

- The PySNMP package must be installed, variously referred to as `pysnmp` or `python-pysnmp`

Enabling the SNMP Hardware Type

1. Add `snmp` to the list of `enabled_hardware_types` in `ironic.conf`. Also update `enabled_management_interfaces` and `enabled_power_interfaces` in `ironic.conf` as shown below:

```
[DEFAULT]
enabled_hardware_types = snmp
enabled_management_interfaces = noop
enabled_power_interfaces = snmp
```

2. To enable the network boot fallback, update `enable_netboot_fallback` in `ironic.conf`:

```
[pxe]
enable_netboot_fallback = True
```

Note: It is important to enable the fallback as SNMP hardware type does not support setting of boot devices. When booting in legacy (BIOS) mode, the generated network booting artifact will force booting from local disk. In UEFI mode, Ironic will configure the boot order using UEFI variables.

3. Restart the Ironic conductor service.

```
service ironic-conductor restart
```

Ironic Node Configuration

Nodes configured to use the SNMP hardware type should have the `driver` field set to the hardware type `snmp`.

The following property values have to be added to the nodes `driver_info` field:

- `snmp_driver`: PDU manufacturer driver name or `auto` to automatically choose ironic snmp driver based on `SNMPv2-MIB::sysObjectID` value as reported by PDU.
- `snmp_address`: the IPv4 address of the PDU controlling this node.
- `snmp_port`: (optional) A non-standard UDP port to use for SNMP operations. If not specified, the default port (161) is used.
- `snmp_outlet`: The power outlet on the PDU (1-based indexing).
- `snmp_version`: (optional) SNMP protocol version (permitted values `1`, `2c` or `3`). If not specified, SNMPv1 is chosen.
- `snmp_community`: (Required for SNMPv1/SNMPv2c unless `snmp_community_read` and/or `snmp_community_write` properties are present in which case the latter take over) SNMP community name parameter for reads and writes to the PDU.
- `snmp_community_read`: SNMP community name parameter for reads to the PDU. Takes precedence over the `snmp_community` property.
- `snmp_community_write`: SNMP community name parameter for writes to the PDU. Takes precedence over the `snmp_community` property.
- `snmp_user`: (Required for SNMPv3) SNMPv3 User-based Security Model (USM) user name. Synonym for now obsolete `snmp_security` parameter.
- `snmp_auth_protocol`: SNMPv3 message authentication protocol ID. Valid values include: `none`, `md5`, `sha` for all pysnmp versions and additionally `sha224`, `sha256`, `sha384`, `sha512` for pysnmp versions 4.4.1 and later. Default is `none` unless `snmp_auth_key` is provided. In the latter case `md5` is the default.
- `snmp_auth_key`: SNMPv3 message authentication key. Must be 8+ characters long. Required when message authentication is used.
- `snmp_priv_protocol`: SNMPv3 message privacy (encryption) protocol ID. Valid values include: `none`, `des`, `3des`, `aes`, `aes192`, `aes256` for all pysnmp version and additionally

`aes192blmt`, `aes256blmt` for pysnmp versions 4.4.3+. Note that message privacy requires using message authentication. Default is `none` unless `snmp_priv_key` is provided. In the latter case `des` is the default.

- `snmp_priv_key`: SNMPv3 message privacy (encryption) key. Must be 8+ characters long. Required when message encryption is used.
- `snmp_context_engine_id`: SNMPv3 context engine ID. Default is the value of authoritative engine ID.
- `snmp_context_name`: SNMPv3 context name. Default is an empty string.

The following command can be used to enroll a node with the `snmp` hardware type:

```
baremetal node create \
--driver snmp --driver-info snmp_driver=<pdu_manufacturer> \
--driver-info snmp_address=<ip_address> \
--driver-info snmp_outlet=<outlet_index> \
--driver-info snmp_community=<community_string>
```

XClarity driver

Overview

The `xclarity` driver is targeted for IMM 2.0 and IMM 3.0 managed Lenovo servers. The xclarity hardware type enables the user to take advantage of XClarity Manager by using the XClarity Python Client.

Prerequisites

- The XClarity Client library should be installed on the ironic conductor node(s).

  For example, it can be installed with `pip`:

  ```
  sudo pip install python-xclarityclient
  ```

Enabling the XClarity driver

1. Add `xclarity` to the list of `enabled_hardware_types`, `enabled_power_interfaces` and `enabled_management_interfaces` in `/etc/ironic/ironic.conf`. For example:

   ```
   [DEFAULT]
   ...
   enabled_hardware_types = ipmi,xclarity
   enabled_power_interfaces = ipmitool,xclarity
   enabled_management_interfaces = ipmitool,xclarity
   ```

2. Restart the ironic conductor service:

```
sudo service ironic-conductor restart

# Or, for RDO:
sudo systemctl restart openstack-ironic-conductor
```

Registering a node with the XClarity driver

Nodes configured to use the driver should have the `driver` property set to `xclarity`.

The following properties are specified in the nodes `driver_info` field and are required:

- `xclarity_manager_ip`: The IP address of the XClarity Controller.
- `xclarity_username`: User account with admin/server-profile access privilege to the XClarity Controller.
- `xclarity_password`: User account password corresponding to the xclarity_username to the XClarity Controller.
- `xclarity_hardware_id`: The hardware ID of the XClarity managed server.

The `baremetal node create` command can be used to enroll a node with the `xclarity` driver. For example:

```
baremetal node create --driver xclarity \
  --driver-info xclarity_manager_ip=https://10.240.217.101 \
  --driver-info xclarity_username=admin \
  --driver-info xclarity_password=password \
  --driver-info xclarity_hardware_id=hardware_id
```

For more information about enrolling nodes see *Enrollment* in the install guide.

4.1.3 Changing Hardware Types and Interfaces

Hardware types and interfaces are enabled in the configuration as described in *Enabling drivers and hardware types*. Usually, a hardware type is configured on enrolling as described in *Enrollment*:

```
baremetal node create --driver <hardware type>
```

Any hardware interfaces can be specified on enrollment as well:

```
baremetal node create --driver <hardware type> \
    --deploy-interface direct --<other>-interface <other implementation>
```

For the remaining interfaces the default value is assigned as described in *Defaults for hardware interfaces*. Both the hardware type and the hardware interfaces can be changed later via the node update API.

Changing Hardware Interfaces

Hardware interfaces can be changed by the following command:

```
baremetal node set <NODE> \
    --deploy-interface direct \
    --<other>-interface <other implementation>
```

The modified interfaces must be enabled and compatible with the current nodes hardware type.

Changing Hardware Type

Changing the nodes hardware type can pose a problem. When the `driver` field is updated, the final result must be consistent, that is, the resulting hardware interfaces must be compatible with the new hardware type. This will not work:

```
baremetal node create --name test --driver fake-hardware
baremetal node set test --driver ipmi
```

This is because the `fake-hardware` hardware type defaults to `fake` implementations for some or all interfaces, but the `ipmi` hardware type is not compatible with them. There are three ways to deal with this situation:

1. Provide new values for all incompatible interfaces, for example:

   ```
   baremetal node set test --driver ipmi \
       --boot-interface pxe \
       --deploy-interface direct \
       --management-interface ipmitool \
       --power-interface ipmitool
   ```

2. Request resetting some of the interfaces to their new defaults by using the `--reset-<IFACE>-interface` family of arguments, for example:

   ```
   baremetal node set test --driver ipmi \
       --reset-boot-interface \
       --reset-deploy-interface \
       --reset-management-interface \
       --reset-power-interface
   ```

   Note: This feature is available starting with ironic 11.1.0 (Rocky series, API version 1.45).

3. Request resetting all interfaces to their new defaults:

   ```
   baremetal node set test --driver ipmi --reset-interfaces
   ```

   You can still specify explicit values for some interfaces:

   ```
   baremetal node set test --driver ipmi --reset-interfaces \
       --deploy-interface direct
   ```

Note: This feature is available starting with ironic 11.1.0 (Rocky series, API version 1.45).

Static boot order configuration

Some hardware is known to misbehave when changing the boot device through the BMC. To work around it you can use the `noop` management interface implementation with the `ipmi` and `redfish` hardware types. In this case the Bare Metal service will not change the boot device for you, leaving the pre-configured boot order.

For example, in case of the *PXE boot*:

1. Via any available means configure the boot order on the node as follows:
    1. Boot from PXE/iPXE on the provisioning NIC.

       Warning: If it is not possible to limit network boot to only provisioning NIC, make sure that no other DHCP/PXE servers are accessible by the node.

    2. Boot from hard drive.
2. Make sure the `noop` management interface is enabled, for example:

```
[DEFAULT]
enabled_hardware_types = ipmi,redfish
enabled_management_interfaces = ipmitool,redfish,noop
```

3. Change the node to use the `noop` management interface:

```
baremetal node set <NODE> --management-interface noop
```

4.1.4 Unsupported drivers

The following drivers were declared as unsupported in ironic Newton release and as of Ocata release they are removed from ironic:

- AMT driver - available as part of ironic-staging-drivers
- iBoot driver - available as part of ironic-staging-drivers
- Wake-On-Lan driver - available as part of ironic-staging-drivers
- Virtualbox drivers
- SeaMicro drivers
- MSFT OCS drivers

The SSH drivers were removed in the Pike release. Similar functionality can be achieved either with VirtualBMC or using libvirt drivers from ironic-staging-drivers.

4.2 Administrators Guide

If you are a system administrator running Ironic, this section contains information that may help you understand how to operate and upgrade the services.

4.2.1 Ironic Python Agent

Overview

Ironic Python Agent (also often called *IPA* or just *agent*) is a Python-based agent which handles *ironic* bare metal nodes in a variety of actions such as inspect, configure, clean and deploy images. IPA is distributed over nodes and runs, inside of a ramdisk, the process of booting this ramdisk on the node.

For more information see the ironic-python-agent documentation.

Drivers

Starting with the Kilo release all deploy interfaces (except for fake ones) are using IPA. For nodes using the *Direct deploy* interface, the conductor prepares a swift temporary URL or a local HTTP URL for the image. IPA then handles the whole deployment process: downloading an image from swift, putting it on the machine and doing any post-deploy actions.

Requirements

Using IPA requires it to be present and configured on the deploy ramdisk, see *Building or downloading a deploy ramdisk image*

Using proxies for image download

Overview

When using the *Direct deploy*, IPA supports using proxies for downloading the user image. For example, this could be used to speed up download by using a caching proxy.

Steps to enable proxies

1. Configure the proxy server of your choice (for example Squid, Apache Traffic Server). This will probably require you to configure the proxy server to cache the content even if the requested URL contains a query, and to raise the maximum cached file size as images can be pretty big. If you have HTTPS enabled in swift (see swift deployment guide), it is possible to configure the proxy server to talk to swift via HTTPS to download the image, store it in the cache unencrypted and return it to the node via HTTPS again. Because the image will be stored unencrypted in the cache, this approach is recommended for images that do not contain sensitive information. Refer to your proxy servers documentation to complete this step.
2. Set `[glance]swift_temp_url_cache_enabled` in the ironic conductor config file to `True`. The conductor will reuse the cached swift temporary URLs instead of generating new ones each time an image is requested, so that the proxy server does not create new cache entries for the same image,

based on the query part of the URL (as it contains some query parameters that change each time it is regenerated).

3. Set `[glance]swift_temp_url_expected_download_start_delay` option in the ironic conductor config file to the value appropriate for your hardware. This is the delay (in seconds) from the time of the deploy request (when the swift temporary URL is generated) to when the URL is used for the image download. You can think of it as roughly the time needed for IPA ramdisk to startup and begin download. This value is used to check if the swift temporary URL duration is large enough to let the image download begin. Also if temporary URL caching is enabled, this will determine if a cached entry will still be valid when the download starts. It is used only if `[glance]swift_temp_url_cache_enabled` is `True`.
4. Increase `[glance]swift_temp_url_duration` option in the ironic conductor config file, as only non-expired links to images will be returned from the swift temporary URLs cache. This means that if `swift_temp_url_duration=1200` then after 20 minutes a new image will be cached by the proxy server as the query in its URL will change. The value of this option must be greater than or equal to `[glance]swift_temp_url_expected_download_start_delay`.
5. Add one or more of `image_http_proxy`, `image_https_proxy`, `image_no_proxy` to driver_info properties in each node that will use the proxy.

Advanced configuration

Out-of-band vs. in-band power off on deploy

After deploying an image onto the nodes hard disk, Ironic will reboot the machine into the new image. By default this power action happens `in-band`, meaning that the ironic-conductor will instruct the IPA ramdisk to power itself off.

Some hardware may have a problem with the default approach and would require Ironic to talk directly to the management controller to switch the power off and on again. In order to tell Ironic to do that, you have to update the nodes `driver_info` field and set the `deploy_forces_oob_reboot` parameter with the value of **True**. For example, the below command sets this configuration in a specific node:

```
baremetal node set <UUID or name> --driver-info deploy_forces_oob_reboot=True
```

4.2.2 Hardware Inspection

Overview

Inspection allows Bare Metal service to discover required node properties once required `driver_info` fields (for example, IPMI credentials) are set by an operator. Inspection will also create the Bare Metal service ports for the discovered ethernet MACs. Operators will have to manually delete the Bare Metal service ports for which physical media is not connected. This is required due to the bug 1405131.

There are two kinds of inspection supported by Bare Metal service:

1. Out-of-band inspection is currently implemented by several hardware types, including `ilo`, `idrac` and `irmc`.
2. *In-band inspection* by utilizing the ironic-inspector project.

The node should be in the `manageable` state before inspection is initiated. If it is in the `enroll` or `available` state, move it to `manageable` first:

```
baremetal node manage <node_UUID>
```

Then inspection can be initiated using the following command:

```
baremetal node inspect <node_UUID>
```

Capabilities discovery

This is an incomplete list of capabilities we want to discover during inspection. The exact support is hardware and hardware type specific though, the most complete list is provided by the iLO *Hardware Inspection Support*.

`secure_boot` (`true` or `false`) whether secure boot is supported for the node

`boot_mode` (`bios` or `uefi`) the boot mode the node is using

`cpu_vt` (`true` or `false`) whether the CPU virtualization is enabled

`cpu_aes` (`true` or `false`) whether the AES CPU extensions are enabled

`max_raid_level` (integer, 0-10) maximum RAID level supported by the node

`pci_gpu_devices` (non-negative integer) number of GPU devices on the node

The operator can specify these capabilities in nova flavor for node to be selected for scheduling:

```
nova flavor-key my-baremetal-flavor set capabilities:pci_gpu_devices="> 0"

nova flavor-key my-baremetal-flavor set capabilities:secure_boot="true"
```

Please see a specific *hardware type page* for the exact list of capabilities this hardware type can discover.

In-band inspection

In-band inspection involves booting a ramdisk on the target node and fetching information directly from it. This process is more fragile and time-consuming than the out-of-band inspection, but it is not vendor-specific and works across a wide range of hardware. In-band inspection is using the ironic-inspector project.

It is supported by all hardware types, and used by default, if enabled, by the `ipmi` hardware type. The `inspector` *inspect* interface has to be enabled to use it:

```
[DEFAULT]
enabled_inspect_interfaces = inspector,no-inspect
```

If the ironic-inspector service is not registered in the service catalog, set the following option:

```
[inspector]
endpoint_override = http://inspector.example.com:5050
```

In order to ensure that ports in Bare Metal service are synchronized with NIC ports on the node, the following settings in the ironic-inspector configuration file must be set:

```
[processing]
add_ports = all
keep_ports = present
```

There are two modes of in-band inspection: *managed inspection* and *unmanaged inspection*.

Managed inspection

Inspection is *managed* when the Bare Metal conductor fully configures the node for inspection, including setting boot device, boot mode and power state. This is the only way to conduct inspection using *Virtual media boot* or with *Layer 3 or DHCP-less ramdisk booting*. This mode is engaged automatically when the node has sufficient information to configure boot (e.g. ports in case of iPXE).

There are a few configuration options that tune managed inspection, the most important is `extra_kernel_params`, which allows adding kernel parameters for inspection specifically. This is where you can configure inspection collectors and other parameters, for example:

```
[inspector]
extra_kernel_params = ipa-inspection-collectors=default,logs ipa-collect-
↪lldp=1
```

For the callback URL the ironic-inspector endpoint from the service catalog is used. If you want to override the endpoint for callback only, set the following option:

```
[inspector]
callback_endpoint_override = https://example.com/baremetal-introspection/v1/
↪continue
```

Unmanaged inspection

Under *unmanaged* inspection we understand in-band inspection orchestrated by ironic-inspector or a third party. This was the only inspection mode before the Ussuri release, and it is still used when the nodes boot cannot be configured by the conductor. The options described above do not affect unmanaged inspection. See ironic-inspector installation guide for more information.

If you want to **prevent** unmanaged inspection from working, set this option:

```
[inspector]
require_managed_boot = True
```

4.2.3 Node Deployment

Contents

Overview

Node deployment is performed by the Bare Metal service to prepare a node for use by a workload. The exact work flow used depends on a number of factors, including the hardware type and interfaces assigned to a node.

Deploy Steps

The Bare Metal service implements deployment by collecting a list of deploy steps to perform on a node from the Power, Deploy, Management, BIOS, and RAID interfaces of the driver assigned to the node. These steps are then ordered by priority and executed on the node when the node is moved to the `deploying` state.

Nodes move to the `deploying` state when attempting to move to the `active` state (when the hardware is prepared for use by a workload). For a full understanding of all state transitions into deployment, please see *Bare Metal State Machine*.

The Bare Metal service added support for deploy steps in the Rocky release.

Order of execution

Deploy steps are ordered from higher to lower priority, where a larger integer is a higher priority. If the same priority is used by deploy steps on different interfaces, the following resolution order is used: Power, Management, Deploy, BIOS, and RAID interfaces.

Agent steps

All deploy interfaces based on ironic-python-agent (i.e. `direct`, `ansible` and any derivatives) expose the following deploy steps:

`deploy.deploy` (priority 100) In this step the node is booted using a provisioning image.

`deploy.write_image` (priority 80) An out-of-band (`ansible`) or in-band (`direct`) step that downloads and writes the image to the node.

`deploy.tear_down_agent` (priority 40) In this step the provisioning image is shut down.

`deploy.switch_to_tenant_network` (priority 30) In this step networking for the node is switched from provisioning to tenant networks.

`deploy.boot_instance` (priority 20) In this step the node is booted into the user image.

Additionally, the `direct` deploy interfaces has:

`deploy.prepare_instance_boot` (priority 60) In this step the boot device is configured and the bootloader is installed.

> **Note:** For the `ansible` deploy interface these steps are done in `deploy.write_image`.

Accordingly, the following priority ranges can be used for custom deploy steps:

> 100 Out-of-band steps to run before deployment.

81 to 99 In-band deploy steps to run before the image is written.

61 to 79 In-band deploy steps to run after the image is written but before the bootloader is installed.

41 to 59 In-band steps to run after the image is written the bootloader is installed.

21 to 39 Out-of-band steps to run after the provisioning image is shut down.

1 to 19 Any steps that are run when the user instance is already running.

In-band steps

More deploy steps can be provided by the ramdisk, see IPA hardware managers documentation for a listing.

Requesting steps

Starting with Bare Metal API version 1.69 user can optionally supply deploy steps for node deployment when invoking deployment or rebuilding. Overlapping steps will take precedence over *Agent steps* and *Deploy Templates* steps.

Using baremetal client deploy steps can be passed via `--deploy-steps` argument. The argument `--deploy-steps` is one of:

- a JSON string
- path to a JSON file whose contents are passed to the API
- -, to read from stdin. This allows piping in the deploy steps.

An example by passing a JSON string:

```
baremetal node deploy <node> \
    --deploy-steps '[{"interface": "bios", "step": "apply_configuration",
↪"args": {"settings": [{"name": "LogicalProc", "value": "Enabled"}]},
↪"priority": 150}]'
```

Format of JSON for deploy steps argument is described in *Deploy step format* section.

> **Note:** Starting with *ironicclient* 4.6.0 you can provide a YAML file for `--deploy-steps`.

Excluding the default steps

Starting with the Xena release, you can use the new *Custom agent deploy* interface to exclude the default step `write_image` and skip bootloader installation in the `prepare_instance_boot` step.

Writing a Deploy Step

Please refer to *Developing deploy and clean steps*.

FAQ

What deploy step is running?

To check what deploy step the node is performing or attempted to perform and failed, run the following command; it will return the value in the nodes `driver_internal_info` field:

```
baremetal node show <node> -f value -c driver_internal_info
```

The `deploy_steps` field will contain a list of all remaining steps with their priorities, and the first one listed is the step currently in progress or that the node failed before going into `deploy failed` state.

Troubleshooting

If deployment fails on a node, the node will be put into the `deploy failed` state until the node is deprovisioned. A deprovisioned node is moved to the `available` state after the cleaning process has been performed successfully.

Strategies for determining why a deploy step failed include checking the ironic conductor logs, checking logs from the ironic-python-agent that have been stored on the ironic conductor, or performing general hardware troubleshooting on the node.

Deploy Templates

Starting with the Stein release, with Bare Metal API version 1.55, deploy templates offer a way to define a set of one or more deploy steps to be executed with particular sets of arguments and priorities.

Each deploy template has a name, which must be a valid trait. Traits can be either standard or custom. Standard traits are listed in the os_traits library. Custom traits must meet the following requirements:

- prefixed with `CUSTOM_`
- contain only upper case characters A to Z, digits 0 to 9, or underscores
- no longer than 255 characters in length

Deploy step format

An invocation of a deploy step is defined in a deploy template as follows:

```
{
    "interface": "<name of the driver interface>",
    "step": "<name of the step>",
    "args": {
        "<arg1>": "<value1>",
        "<arg2>": "<value2>"
    },
    "priority": <priority of the step>
}
```

A deploy template contains a list of one or more such steps. Each combination of *interface* and *step* may only be specified once in a deploy template.

Matching deploy templates

During deployment, if any of the traits in a nodes `instance_info.traits` field match the name of a deploy template, then the steps from that deploy template will be added to the list of steps to be executed by the node.

When using the Compute service, any traits in the instances flavor properties or image properties are stored in `instance_info.traits` during deployment. See *Scheduling based on traits* for further information on how traits are used for scheduling when the Bare Metal service is used with the Compute service.

Note that there is no ongoing relationship between a node and any templates that are matched during deployment. The set of matching deploy templates is checked at deployment time. Any subsequent updates to or deletion of those templates will not be reflected in the nodes configuration unless it is redeployed or rebuilt. Similarly, if a node is rebuilt and the set of matching deploy templates has changed since the initial deployment, then the resulting configuration of the node may be different from the initial deployment.

Overriding default deploy steps

A deploy step is enabled by default if it has a non-zero default priority. A default deploy step may be overridden in a deploy template. If the steps priority is a positive integer it will be executed with the specified priority and arguments. If the steps priority is zero, the step will not be executed.

If the *deploy.deploy step* is included in a deploy template, it can only be assigned a priority of zero to disable it.

Creating a deploy template via API

A deploy template can be created using the Bare Metal API:

```
POST /v1/deploy_templates
```

Here is an example of the body of a request to create a deploy template with a single step:

```
{
    "name": "CUSTOM_HYPERTHREADING_ON",
    "steps": [
        {
            "interface": "bios",
            "step": "apply_configuration",
            "args": {
                "settings": [
                    {
                        "name": "LogicalProc",
                        "value": "Enabled"
                    }
                ]
            },
            "priority": 150
        }
    ]
}
```

Further information on this API is available here.

Creating a deploy template via baremetal client

A deploy template can be created via the `baremetal deploy template create` command, starting with `python-ironicclient` 2.7.0.

The argument `--steps` must be specified. Its value is one of:

- a JSON string
- path to a JSON file whose contents are passed to the API
- -, to read from stdin. This allows piping in the deploy steps.

Example of creating a deploy template with a single step using a JSON string:

```
baremetal deploy template create \
    CUSTOM_HYPERTHREADING_ON \
    --steps '[{"interface": "bios", "step": "apply_configuration", "args": {
↪"settings": [{"name": "LogicalProc", "value": "Enabled"}]}, "priority": 150}
↪]'
```

Or with a file:

```
baremetal deploy template create \
    CUSTOM_HYPERTHREADING_ON \
    ---steps my-deploy-steps.txt
```

Or with stdin:

```
cat my-deploy-steps.txt | baremetal deploy template create \
    CUSTOM_HYPERTHREADING_ON \
    --steps -
```

Example of use with the Compute service

Note: The deploy steps used in this example are for example purposes only.

In the following example, we first add the trait `CUSTOM_HYPERTHREADING_ON` to the node represented by <node>:

```
baremetal node add trait <node> CUSTOM_HYPERTHREADING_ON
```

We also update the flavor `bm-hyperthreading-on` in the Compute service with the following property:

```
openstack flavor set --property trait:CUSTOM_HYPERTHREADING_ON=required bm-
↪hyperthreading-on
```

Creating a Compute instance with this flavor will ensure that the instance is scheduled only to Bare Metal nodes with the `CUSTOM_HYPERTHREADING_ON` trait.

We could then create a Bare Metal deploy template with the name `CUSTOM_HYPERTHREADING_ON` and a deploy step that enables Hyperthreading:

```
{
    "name": "CUSTOM_HYPERTHREADING_ON",
    "steps": [
        {
            "interface": "bios",
            "step": "apply_configuration",
            "args": {
                "settings": [
                    {
                        "name": "LogicalProc",
                        "value": "Enabled"
                    }
                ]
            },
            "priority": 150
        }
    ]
}
```

When an instance is created using the `bm-hyperthreading-on` flavor, then the deploy steps of deploy template `CUSTOM_HYPERTHREADING_ON` will be executed during the deployment of the scheduled node, causing Hyperthreading to be enabled in the nodes BIOS configuration.

To make this example more dynamic, lets add a second trait `CUSTOM_HYPERTHREADING_OFF` to the node:

```
baremetal node add trait <node> CUSTOM_HYPERTHREADING_OFF
```

We could also update a second flavor, `bm-hyperthreading-off`, with the following property:

```
openstack flavor set --property trait:CUSTOM_HYPERTHREADING_OFF=required bm-
↪hyperthreading-off
```

Finally, we create a deploy template with the name `CUSTOM_HYPERTHREADING_OFF` and a deploy step that disables Hyperthreading:

```
{
    "name": "CUSTOM_HYPERTHREADING_OFF",
    "steps": [
        {
            "interface": "bios",
            "step": "apply_configuration",
            "args": {
                "settings": [
                    {
                        "name": "LogicalProc",
                        "value": "Disabled"
                    }
                ]
            },
            "priority": 150
        }
    ]
}
```

Creating a Compute instance with the `bm-hyperthreading-off` instance will cause the scheduled node to have Hyperthreading disabled in the BIOS during deployment.

We now have a way to create Compute instances with different configurations, by choosing between different Compute flavors, supported by a single Bare Metal node that is dynamically configured during deployment.

4.2.4 Node cleaning

Overview

Ironic provides two modes for node cleaning: `automated` and `manual`.

`Automated cleaning` is automatically performed before the first workload has been assigned to a node and when hardware is recycled from one workload to another.

`Manual cleaning` must be invoked by the operator.

Automated cleaning

When hardware is recycled from one workload to another, ironic performs automated cleaning on the node to ensure its ready for another workload. This ensures the tenant will get a consistent bare metal node deployed every time.

Ironic implements automated cleaning by collecting a list of cleaning steps to perform on a node from the Power, Deploy, Management, BIOS, and RAID interfaces of the driver assigned to the node. These steps are then ordered by priority and executed on the node when the node is moved to `cleaning` state, if automated cleaning is enabled.

With automated cleaning, nodes move to `cleaning` state when moving from `active -> available` state (when the hardware is recycled from one workload to another). Nodes also traverse cleaning when going from `manageable -> available` state (before the first workload is assigned to the nodes). For a full understanding of all state transitions into cleaning, please see *Bare Metal State Machine*.

Ironic added support for automated cleaning in the Kilo release.

Enabling automated cleaning

To enable automated cleaning, ensure that your ironic.conf is set as follows:

```
[conductor]
automated_clean=true
```

This will enable the default set of cleaning steps, based on your hardware and ironic hardware types used for nodes. This includes, by default, erasing all of the previous tenants data.

You may also need to configure a *Cleaning Network*.

Cleaning steps

Cleaning steps used for automated cleaning are ordered from higher to lower priority, where a larger integer is a higher priority. In case of a conflict between priorities across interfaces, the following resolution order is used: Power, Management, Deploy, BIOS, and RAID interfaces.

You can skip a cleaning step by setting the priority for that cleaning step to zero or None.

You can reorder the cleaning steps by modifying the integer priorities of the cleaning steps.

See *How do I change the priority of a cleaning step?* for more information.

Storage cleaning options

Clean steps specific to storage are `erase_devices`, `erase_devices_metadata` and (added in Yoga) `erase_devices_express`.

`erase_devices` aims to ensure that the data is removed in the most secure way available. On devices that support hardware assisted secure erasure (many NVMe and some ATA drives) this is the preferred option. If hardware-assisted secure erasure is not available and if `[deploy]/continue_if_disk_secure_erase_fails` is set to `True`, cleaning will fall back to using `shred` to overwrite the contents of the device. Otherwise cleaning will fail. It is important to note that `erase_devices` may take a very long time (hours or even days) to complete, unless fast, hardware

assisted data erasure is supported by all the devices in a system. Generally, it is very difficult (if possible at all) to recover data after performing cleaning with `erase_devices`.

`erase_devices_metadata` clean step doesnt provide as strong assurance of irreversible destruction of data as `erase_devices`. However, it has the advantage of a reasonably quick runtime (seconds to minutes). It operates by destroying metadata of the storage device without erasing every bit of the data itself. Attempts of restoring data after running `erase_devices_metadata` may be successful but would certainly require relevant expertise and specialized tools.

Lastly, `erase_devices_express` combines some of the perks of both `erase_devices` and `erase_devices_metadata`. It attempts to utilize hardware assisted data erasure features if available (currently only NVMe devices are supported). In case hardware-asssisted data erasure is not available, it falls back to metadata erasure for the device (which is identical to `erase_devices_metadata`). It can be considered a time optimized mode of storage cleaning, aiming to perform as thorough data erasure as it is possible within a short period of time. This clean step is particularly well suited for environments with hybrid NVMe-HDD storage configuration as it allows fast and secure erasure of data stored on NVMes combined with equally fast but more basic metadata-based erasure of data on HDDs. `erase_devices_express` is disabled by default. In order to use it, the following configuration is recommended.

```
[deploy]/erase_devices_priority=0
[deploy]/erase_devices_metadata_priority=0
[conductor]/clean_step_priority_override=deploy.erase_devices_express:5
```

This ensures that `erase_devices` and `erase_devices_metadata` are disabled so that storage is not cleaned twice and then assigns a non-zero priority to `erase_devices_express`, hence enabling it. Any non-zero priority specified in the priority override will work.

Also *[deploy]/enable_nvme_secure_erase* should not be disabled (it is on by default).

Management Interface

Table 5: idrac cleaning steps

Name	Details	Priority	Stoppable	Arguments
`clear_job_queue`	Clear the job queue.	0	no	
`known_good_state`	Reset the iDRAC, Clear the job queue.	0	no	
`reset_idrac`	Reset the iDRAC.	0	no	

Table 6: idrac-redfish cleaning steps

Name	Details	Priority	Stoppable	Arguments
`clear_job_queue`	Clear iDRAC job queue.	0	no	
`clear_secure_boot_keys`	Clear all secure boot keys.	0	no	
`export_configuration`	Export the configuration of the server. Exports the configuration of the server against which the step is run and stores it in specific format in indicated location. Uses Dells Server Configuration Profile (SCP) from *sushy-oem-idrac* library to get ALL configuration for cloning. **param task** A task from TaskManager. **param export_configuration_location** URL of location to save the configuration to.	0	no	`export_configuration_location` (*required*) URL of location to save the configuration to.
`import_configuration`	Import and apply the configuration to the server. Gets pre-created configuration from storage by given location and imports that into given server. Uses Dells Server Configuration Profile (SCP). **param task** A task from TaskManager. **param import_configuration_location** URL of location to fetch desired configuration from.	0	no	`import_configuration_location` (*required*) URL of location to fetch desired configuration from.
`import_export_configuration`	Import and export configuration in one go. Gets pre-created configuration from storage by given name and imports that into given server. After that ex-	0	no	`export_configuration_location` (*required*) URL of location to save the configuration to. `import_configuration_location` (*required*) URL of location to fetch desired configura-

Table 7: idrac-wsman cleaning steps

Name	Details	Priority	Stoppable	Arguments
`clear_job_queue`	Clear the job queue.	0	no	
`known_good_state`	Reset the iDRAC, Clear the job queue.	0	no	
`reset_idrac`	Reset the iDRAC.	0	no	

Table 8: ilo cleaning steps

Name	Details	Priority	Stoppable	Arguments
`activate_license`	Activates iLO Advanced license.	0	no	`ilo_license_key` (*required*) The HPE iLO Advanced license key to activate enterprise features.
`clear_secure_boot_keys`	Clear all secure boot keys. Clears all the secure boot keys. This operation is supported only on HP Proliant Gen9 and above servers.	0	no	
`reset_bios_to_default`	Reset the BIOS settings to default values. Resets BIOS to default settings. This operation is currently supported only on HP Proliant Gen9 and above servers.	10	no	
`reset_ilo`	Resets the iLO.	0	no	
`reset_ilo_credential`	Resets the iLO password.	30	no	
`reset_secure_boot_keys_to_default`	Reset secure boot keys to manufacturing defaults. Resets the secure boot keys to manufacturing defaults. This operation is supported only on HP Proliant Gen9 and above servers.	20	no	
`security_parameters_update`	Update the security parameters.	0	no	`security_parameters` (*required*) This argument represents the ordered list of JSON dictionaries of security parameters. Each security parameter consists of three fields, namely param, ignore and enable from which param field will be mandatory. These fields represent security parameter name, ignore flag and state of the security parameter. The supported security parameter names are password_complexity, require_login_for_ilo_rbsu, ipmi_over_lan, secure_boot, require_host_authentication. The security parameters will be updated (in the order given) one by one on the baremetal server.
`update_auth_failure_logging_threshold`	Update Authentication Failure Logging Threshold security parameter.	0	no	`ignore` This argument represents boolean parameter. If set True the security parameters will be ignored by

Table 9: ilo5 cleaning steps

Name	Details	Priority	Stoppable	Arguments
`activate_license`	Activates iLO Advanced license.	0	no	`ilo_license_key` (*required*) The HPE iLO Advanced license key to activate enterprise features.
`clear_ca_certificates`	Clears the certificates provided in the list of files to iLO.	0	no	`certificate_files` (*required*) The list of files containing the certificates to be cleared. If empty list is specified, all the certificates on the ilo will be cleared, except the certificates in the file configured with configuration parameter webserver_verify_ca are spared as they are required for booting the deploy image for some boot interfaces.
`clear_secure_boot_keys`	Clear all secure boot keys. Clears all the secure boot keys. This operation is supported only on HP Proliant Gen9 and above servers.	0	no	
`erase_devices`	Erase all the drives on the node. This method performs out-of-band sanitize disk erase on all the supported physical drives in the node. This erase cannot be performed on logical drives.	0	no	`erase_pattern` Dictionary of disk type and corresponding erase pattern to be used to perform specific out-of-band sanitize disk erase. Supported values are, for hdd: (overwrite, crypto, zero), for ssd: (block, crypto, zero). Default pattern is: {hdd: overwrite, ssd: block}.
`one_button_secure_erase`	Erase the whole system securely. The One-button secure erase process resets iLO and deletes all licenses stored there, resets BIOS settings, and deletes all Active Health System (AHS) and warranty data stored on the system. It also erases supported non-volatile storage data and deletes any deployment setting profiles.	0	no	
`reset_bios_to_default`	Reset the BIOS settings to default values. Resets BIOS to default settings. This operation is currently supported only on HP Proliant Gen9 and above servers.	10	no	

Table 10: irmc cleaning steps

Name	Details	Priority	Stoppable	Arguments
`restore_irmc_bios_config`	Restore BIOS config for a node.	0	no	

Table 11: redfish cleaning steps

Name	Details	Priority	Stoppable	Arguments
`clear_secure_boot_keys`	Clear all secure boot keys.	0	no	
`reset_secure_boot_keys_to_default`	Reset secure boot keys to manufacturing defaults.	0	no	
`update_firmware`	Updates the firmware on the node.	0	no	`firmware_images` (*required*) A list of firmware images to apply.

Bios Interface

Table 12: idrac-redfish cleaning steps

Name	Details	Priority	Stoppable	Arguments
`apply_configuration`	Apply the BIOS settings to the node.	0	no	`settings` (*required*) A list of BIOS settings to be applied
`factory_reset`	Reset the BIOS settings of the node to the factory default.	0	no	

Table 13: idrac-wsman cleaning steps

Name	Details	Priority	Stoppable	Arguments
`apply_configuration`	Apply the BIOS configuration to the node **param task** a TaskManager instance containing the node to act on **param settings** List of BIOS settings to apply **raises** DRACOperationError upon an error from pythondracclient	0	no	`settings` (*required*) List of BIOS settings to apply
`factory_reset`	Reset the BIOS settings of the node to the factory default. This uses the Lifecycle Controller configuration to perform BIOS configuration reset. Leveraging the pythondracclient methods already available.	0	no	

Table 14: ilo cleaning steps

Name	Details	Priority	Stoppable	Arguments
`apply_configuration`	Applies the provided configuration on the node.	0	no	`settings` (*required*) Dictionary with current BIOS configuration.
`factory_reset`	Reset the BIOS settings to factory configuration.	0	no	

Table 15: irmc cleaning steps

Name	Details	Priority	Stoppable	Arguments
`apply_configuration`	Applies BIOS configuration on the given node. This method takes the BIOS settings from the settings param and applies BIOS configuration on the given node. After the BIOS configuration is done, self.cache_bios_settings() may be called to sync the nodes BIOS-related information with the BIOS configuration applied on the node. It will also validate the given settings before applying any settings and manage failures when setting an invalid BIOS config. In the case of needing password to update the BIOS config, it will be taken from the driver_info properties.	0	no	`settings` (*required*) Dictionary containing the BIOS configuration.

Table 16: redfish cleaning steps

Name	Details	Priority	Stoppable	Arguments
`apply_configuration`	Apply the BIOS settings to the node.	0	no	`settings` (*required*) A list of BIOS settings to be applied
`factory_reset`	Reset the BIOS settings of the node to the factory default.	0	no	

Raid Interface

Table 17: agent cleaning steps

Name	Details	Priority	Stoppable	Arguments
`create_configuration`	Create a RAID configuration on a bare metal using agent ramdisk. This method creates a RAID configuration on the given node.	0	no	
`delete_configuration`	Deletes RAID configuration on the given node.	0	no	

Table 18: ibmc cleaning steps

Name	Details	Priority	Stoppable	Arguments
`create_configuration`	Create a RAID configuration. This method creates a RAID configuration on the given node.	0	no	`create_nonroot_volumes` This specifies whether to create the non-root volumes. Defaults to *True*. `create_root_volume` This specifies whether to create the root volume. Defaults to *True*. `delete_existing` Setting this to True indicates to delete existing RAID configuration prior to creating the new configuration. Default value is False.
`delete_configuration`	Delete the RAID configuration.	0	no	

Table 19: idrac cleaning steps

Name	Details	Priority	Stoppable	Arguments
`create_configuration`	Create the RAID configuration. This method creates the RAID configuration on the given node.	0	no	`create_nonroot_volumes` This specifies whether to create the non-root volumes. Defaults to *True*. `create_root_volume` This specifies whether to create the root volume. Defaults to *True*. `delete_existing` Setting this to True indicates to delete existing RAID configuration prior to creating the new configuration. Default value is False.
`delete_configuration`	Delete the RAID configuration.	0	no	

Table 20: idrac-redfish cleaning steps

Name	Details	Priority	Stoppable	Arguments
`create_configuration`	Create RAID configuration on the node. This method creates the RAID configuration as read from node.target_raid_config. This method by default will create all logical disks.	0	no	`create_nonroot_volumes` This specifies whether to create the non-root volumes. Defaults to *True*. `create_root_volume` This specifies whether to create the root volume. Defaults to *True*. `delete_existing` Setting this to *True* indicates to delete existing RAID configuration prior to creating the new configuration. Default value is *False*.
`delete_configuration`	Delete RAID configuration on the node.	0	no	

Table 21: idrac-wsman cleaning steps

Name	Details	Priority	Stoppable	Arguments
`create_configuration`	Create the RAID configuration. This method creates the RAID configuration on the given node.	0	no	`create_nonroot_volumes` This specifies whether to create the non-root volumes. Defaults to *True*. `create_root_volume` This specifies whether to create the root volume. Defaults to *True*. `delete_existing` Setting this to True indicates to delete existing RAID configuration prior to creating the new configuration. Default value is False.
`delete_configuration`	Delete the RAID configuration.	0	no	

Table 22: ilo5 cleaning steps

Name	Details	Priority	Stoppable	Arguments
`create_configuration`	Create a RAID configuration on a bare metal using agent ramdisk. This method creates a RAID configuration on the given node.	0	no	`create_nonroot_volumes` This specifies whether to create the non-root volumes. Defaults to *True*. `create_root_volume` This specifies whether to create the root volume. Defaults to *True*.
`delete_configuration`	Delete the RAID configuration.	0	no	

Table 23: irmc cleaning steps

Name	Details	Priority	Stoppable	Arguments
`create_configuration`	Create the RAID configuration. This method creates the RAID configuration on the given node.	0	no	`create_nonroot_volumes` This specifies whether to create the non-root volumes. Defaults to *True*. `create_root_volume` This specifies whether to create the root volume.Defaults to *True*.
`delete_configuration`	Delete the RAID configuration.	0	no	

Table 24: redfish cleaning steps

Name	Details	Priority	Stoppable	Arguments
`create_configuration`	Create RAID configuration on the node. This method creates the RAID configuration as read from node.target_raid_config. This method by default will create all logical disks.	0	no	`create_nonroot_volumes` This specifies whether to create the non-root volumes. Defaults to *True*. `create_root_volume` This specifies whether to create the root volume. Defaults to *True*. `delete_existing` Setting this to *True* indicates to delete existing RAID configuration prior to creating the new configuration. Default value is *False*.
`delete_configuration`	Delete RAID configuration on the node.	0	no	

Manual cleaning

`Manual cleaning` is typically used to handle long running, manual, or destructive tasks that an operator wishes to perform either before the first workload has been assigned to a node or between workloads. When initiating a manual clean, the operator specifies the cleaning steps to be performed. Manual cleaning can only be performed when a node is in the `manageable` state. Once the manual cleaning is finished, the node will be put in the `manageable` state again.

Ironic added support for manual cleaning in the 4.4 (Mitaka series) release.

Setup

In order for manual cleaning to work, you may need to configure a *Cleaning Network*.

Starting manual cleaning via API

Manual cleaning can only be performed when a node is in the `manageable` state. The REST API request to initiate it is available in API version 1.15 and higher:

```
PUT /v1/nodes/<node_ident>/states/provision
```

(Additional information is available here.)

This API will allow operators to put a node directly into `cleaning` provision state from `manageable` state via target: clean. The PUT will also require the argument clean_steps to be specified. This is an ordered list of cleaning steps. A cleaning step is represented by a dictionary (JSON), in the form:

```
{
    "interface": "<interface>",
    "step": "<name of cleaning step>",
    "args": {"<arg1>": "<value1>", ..., "<argn>": <valuen>}
}
```

The interface and step keys are required for all steps. If a cleaning step method takes keyword arguments, the args key may be specified. It is a dictionary of keyword variable arguments, with each keyword-argument entry being <name>: <value>.

If any step is missing a required keyword argument, manual cleaning will not be performed and the node will be put in `clean failed` provision state with an appropriate error message.

If, during the cleaning process, a cleaning step determines that it has incorrect keyword arguments, all earlier steps will be performed and then the node will be put in `clean failed` provision state with an appropriate error message.

An example of the request body for this API:

```
{
  "target":"clean",
  "clean_steps": [{
    "interface": "raid",
    "step": "create_configuration",
    "args": {"create_nonroot_volumes": false}
  },
  {
    "interface": "deploy",
    "step": "erase_devices"
  }]
}
```

In the above example, the nodes RAID interface would configure hardware RAID without non-root volumes, and then all devices would be erased (in that order).

Starting manual cleaning via openstack metal CLI

Manual cleaning is available via the `baremetal node clean` command, starting with Bare Metal API version 1.15.

The argument `--clean-steps` must be specified. Its value is one of:

- a JSON string
- path to a JSON file whose contents are passed to the API
- -, to read from stdin. This allows piping in the clean steps. Using - to signify stdin is common in Unix utilities.

The following examples assume that the Bare Metal API version was set via the `OS_BAREMETAL_API_VERSION` environment variable. (The alternative is to add `--os-baremetal-api-version 1.15` to the command.):

```
export OS_BAREMETAL_API_VERSION=1.15
```

Examples of doing this with a JSON string:

```
baremetal node clean <node> \
    --clean-steps '[{"interface": "deploy", "step": "erase_devices_metadata"}]
↪'

baremetal node clean <node> \
    --clean-steps '[{"interface": "deploy", "step": "erase_devices"}]'
```

Or with a file:

```
baremetal node clean <node> \
    --clean-steps my-clean-steps.txt
```

Or with stdin:

```
cat my-clean-steps.txt | baremetal node clean <node> \
    --clean-steps -
```

Cleaning Network

If you are using the Neutron DHCP provider (the default) you will also need to ensure you have configured a cleaning network. This network will be used to boot the ramdisk for in-band cleaning. You can use the same network as your tenant network. For steps to set up the cleaning network, please see *Configure the Bare Metal service for cleaning*.

In-band vs out-of-band

Ironic uses two main methods to perform actions on a node: in-band and out-of-band. Ironic supports using both methods to clean a node.

In-band

In-band steps are performed by ironic making API calls to a ramdisk running on the node using a deploy interface. Currently, all the deploy interfaces support in-band cleaning. By default, ironic-python-agent ships with a minimal cleaning configuration, only erasing disks. However, you can add your own cleaning steps and/or override default cleaning steps with a custom Hardware Manager.

Out-of-band

Out-of-band are actions performed by your management controller, such as IPMI, iLO, or DRAC. Out-of-band steps will be performed by ironic using a power or management interface. Which steps are performed depends on the hardware type and hardware itself.

For Out-of-Band cleaning operations supported by iLO hardware types, refer to *Node Cleaning Support*.

FAQ

How are cleaning steps ordered?

For automated cleaning, cleaning steps are ordered by integer priority, where a larger integer is a higher priority. In case of a conflict between priorities across hardware interfaces, the following resolution order is used:

1. Power interface
2. Management interface
3. Deploy interface
4. BIOS interface
5. RAID interface

For manual cleaning, the cleaning steps should be specified in the desired order.

How do I skip a cleaning step?

For automated cleaning, cleaning steps with a priority of 0 or None are skipped.

How do I change the priority of a cleaning step?

For manual cleaning, specify the cleaning steps in the desired order.

For automated cleaning, it depends on whether the cleaning steps are out-of-band or in-band.

Most out-of-band cleaning steps have an explicit configuration option for priority.

Changing the priority of an in-band (ironic-python-agent) cleaning step requires use of a custom HardwareManager. The only exception is `erase_devices`, which can have its priority set in ironic.conf. For instance, to disable erase_devices, youd set the following configuration option:

```
[deploy]
erase_devices_priority=0
```

To enable/disable the in-band disk erase using `ilo` hardware type, use the following configuration option:

```
[ilo]
clean_priority_erase_devices=0
```

The generic hardware manager first identifies whether a device is an NVMe drive or an ATA drive so that it can attempt a platform-specific secure erase method. In case of NVMe drives, it tries to perform a secure format operation by using the `nvme-cli` utility. This behavior can be controlled using the following configuration option (by default it is set to True):

```
[deploy]
enable_nvme_secure_erase=True
```

In case of ATA drives, it tries to perform ATA disk erase by using the `hdparm` utility.

If neither method is supported, it performs software based disk erase using the `shred` utility. By default, the number of iterations performed by `shred` for software based disk erase is 1. To configure the number of iterations, use the following configuration option:

```
[deploy]
erase_devices_iterations=1
```

Overriding step priority

`[conductor]clean_step_priority_override` is a new configuration option which allows specifying priority of each step using multiple configuration values:

```
[conductor]
clean_step_priority_override=deploy.erase_devices_metadata:123
clean_step_priority_override=management.reset_bios_to_default:234
clean_step_priority_override=management.clean_priority_reset_ilo:345
```

This parameter can be specified as many times as required to define priorities for several cleaning steps - the values will be combined.

What cleaning step is running?

To check what cleaning step the node is performing or attempted to perform and failed, run the following command; it will return the value in the nodes `driver_internal_info` field:

```
baremetal node show $node_ident -f value -c driver_internal_info
```

The `clean_steps` field will contain a list of all remaining steps with their priorities, and the first one listed is the step currently in progress or that the node failed before going into `clean failed` state.

Should I disable automated cleaning?

Automated cleaning is recommended for ironic deployments, however, there are some tradeoffs to having it enabled. For instance, ironic cannot deploy a new instance to a node that is currently cleaning, and cleaning can be a time consuming process. To mitigate this, we suggest using NVMe drives with support for NVMe Secure Erase (based on `nvme-cli` format command) or ATA drives with support for cryptographic ATA Security Erase, as typically the erase_devices step in the deploy interface takes the longest time to complete of all cleaning steps.

Why cant I power on/off a node while its cleaning?

During cleaning, nodes may be performing actions that shouldnt be interrupted, such as BIOS or Firmware updates. As a result, operators are forbidden from changing power state via the ironic API while a node is cleaning.

Troubleshooting

If cleaning fails on a node, the node will be put into `clean failed` state. If the failure happens while running a clean step, the node is also placed in maintenance mode to prevent ironic from taking actions on the node. The operator should validate that no permanent damage has been done to the node and no processes are still running on it before removing the maintenance mode.

Note: Older versions of ironic may put the node to maintenance even when no clean step has been running.

Nodes in `clean failed` will not be powered off, as the node might be in a state such that powering it off could damage the node or remove useful information about the nature of the cleaning failure.

A `clean failed` node can be moved to `manageable` state, where it cannot be scheduled by nova and you can safely attempt to fix the node. To move a node from `clean failed` to `manageable`:

```
baremetal node manage $node_ident
```

You can now take actions on the node, such as replacing a bad disk drive.

Strategies for determining why a cleaning step failed include checking the ironic conductor logs, viewing logs on the still-running ironic-python-agent (if an in-band step failed), or performing general hardware troubleshooting on the node.

When the node is repaired, you can move the node back to `available` state, to allow it to be scheduled by nova.

```
# First, move it out of maintenance mode
baremetal node maintenance unset $node_ident

# Now, make the node available for scheduling by nova
baremetal node provide $node_ident
```

The node will begin automated cleaning from the start, and move to `available` state when complete.

4.2.5 Node adoption

Overview

As part of hardware inventory lifecycle management, it is not an unreasonable need to have the capability to be able to add hardware that should be considered in-use by the Bare Metal service, that may have been deployed by another Bare Metal service installation or deployed via other means.

As such, the node adoption feature allows a user to define a node as `active` while skipping the `available` and `deploying` states, which will prevent the node from being seen by the Compute service as ready for use.

This feature is leveraged as part of the state machine workflow, where a node in `manageable` can be moved to `active` state via the provision_state verb `adopt`. To view the state transition capabilities, please see *Bare Metal State Machine*.

Note: For deployments using Ironic in conjunction with Nova, Ironics node adoption feature is not suitable. If you need to adopt production nodes into Ironic **and** Nova, you can find a high-level recipe in *Adoption with Nova*.

How it works

A node initially enrolled begins in the `enroll` state. An operator must then move the node to `manageable` state, which causes the nodes `power` interface to be validated. Once in `manageable` state, an operator can then explicitly choose to adopt a node.

Adoption of a node results in the validation of its `boot` interface, and upon success the process leverages what is referred to as the takeover logic. The takeover process is intended for conductors to take over the management of nodes for a conductor that has failed.

The takeover process involves the deploy interfaces `prepare` and `take_over` methods being called. These steps take specific actions such as downloading and staging the deployment kernel and ramdisk, ISO image, any required boot image, or boot ISO image and then places any PXE or virtual media configuration necessary for the node should it be required.

The adoption process makes no changes to the physical node, with the exception of operator supplied configurations where virtual media is used to boot the node under normal circumstances. An operator should ensure that any supplied configuration defining the node is sufficient for the continued operation of the node moving forward.

Possible Risk

The main risk with this feature is that supplied configuration may ultimately be incorrect or invalid which could result in potential operational issues:

- `rebuild` verb - Rebuild is intended to allow a user to re-deploy the node to a fresh state. The risk with adoption is that the image defined when an operator adopts the node may not be the valid image for the pre-existing configuration.

  If this feature is utilized for a migration from one deployment to another, and pristine original images are loaded and provided, then ultimately the risk is the same with any normal use of the `rebuild` feature, the server is effectively wiped.

- When deleting a node, the deletion or cleaning processes may fail if the incorrect deployment image is supplied in the configuration as the node may NOT have been deployed with the supplied image and driver or compatibility issues may exist as a result.

  Operators will need to be cognizant of that possibility and should plan accordingly to ensure that deployment images are known to be compatible with the hardware in their environment.

- Networking - Adoption will assert no new networking configuration to the newly adopted node as that would be considered modifying the node.

  Operators will need to plan accordingly and have network configuration such that the nodes will be able to network boot.

How to use

Note: The power state that the ironic-conductor observes upon the first successful power state check, as part of the transition to the `manageable` state will be enforced with a node that has been adopted. This means a node that is in `power off` state will, by default, have the power state enforced as `power off` moving forward, unless an administrator actively changes the power state using the Bare Metal service.

Requirements

Requirements for use are essentially the same as to deploy a node:

- Sufficient driver information to allow for a successful power management validation.
- Sufficient instance_info to pass deploy interface preparation.

Each driver may have additional requirements dependent upon the configuration that is supplied. An example of this would be defining a node to always boot from the network, which will cause the conductor to attempt to retrieve the pertinent files. Inability to do so will result in the adoption failing, and the node being placed in the `adopt failed` state.

Example

This is an example to create a new node, named `testnode`, with sufficient information to pass basic validation in order to be taken from the `manageable` state to `active` state:

```
# Explicitly set the client API version environment variable to
# 1.17, which introduces the adoption capability.
export OS_BAREMETAL_API_VERSION=1.17

baremetal node create --name testnode \
    --driver ipmi \
    --driver-info ipmi_address=<ip_address> \
    --driver-info ipmi_username=<username> \
    --driver-info ipmi_password=<password> \
    --driver-info deploy_kernel=<deploy_kernel_id_or_url> \
    --driver-info deploy_ramdisk=<deploy_ramdisk_id_or_url>

baremetal port create <node_mac_address> --node <node_uuid>

baremetal node set testnode \
    --instance-info image_source="http://localhost:8080/blankimage" \
    --instance-info capabilities="{\"boot_option\": \"local\"}"
```

(continues on next page)

(continued from previous page)

```
baremetal node manage testnode --wait

baremetal node adopt testnode --wait
```

Note: In the above example, the image_source setting must reference a valid image or file, however that image or file can ultimately be empty.

Note: The above example utilizes a capability that defines the boot operation to be local. It is recommended to define the node as such unless network booting is desired.

Note: The above example will fail a re-deployment as a fake image is defined and no instance_info/image_checksum value is defined. As such any actual attempt to write the image out will fail as the image_checksum value is only validated at time of an actual deployment operation.

Note: A user may wish to assign an instance_uuid to a node, which could be used to match an instance in the Compute service. Doing so is not required for the proper operation of the Bare Metal service.

baremetal node set <node name or uuid> instance-uuid <uuid>

Note: In Newton, coupled with API version 1.20, the concept of a network_interface was introduced. A user of this feature may wish to add new nodes with a network_interface of `noop` and then change the interface at a later point and time.

Troubleshooting

Should an adoption operation fail for a node, the error that caused the failure will be logged in the nodes `last_error` field when viewing the node. This error, in the case of node adoption, will largely be due to failure of a validation step. Validation steps are dependent upon what driver is selected for the node.

Any node that is in the `adopt failed` state can have the `adopt` verb re-attempted. Example:

```
baremetal node adopt <node name or uuid>
```

If a user wishes to abort their attempt at adopting, they can then move the node back to `manageable` from `adopt failed` state by issuing the `manage` verb. Example:

```
baremetal node manage <node name or uuid>
```

If all else fails the hardware node can be removed from the Bare Metal service. The `node delete` command, which is **not** the same as setting the provision state to `deleted`, can be used while the node is in `adopt failed` state. This will delete the node without cleaning occurring to preserve the nodes current state. Example:

```
baremetal node delete <node name or uuid>
```

Adoption with Nova

Since there is no mechanism to create bare metal instances in Nova when nodes are adopted into Ironic, the node adoption feature described above cannot be used to add in production nodes to deployments which use Ironic together with Nova.

One option to add in production nodes to an Ironic/Nova deployment is to use the fake drivers. The overall idea is that for Nova the nodes are instantiated normally to ensure the instances are properly created in the compute project while Ironic does not touch them.

Here are some high level steps to be used as a guideline:

- create a bare metal flavor and a hosting project for the instances
- enroll the nodes into Ironic, create the ports, move them to manageable
- change the hardware type and the interfaces to fake drivers
- provide the nodes to make them available
- one by one, add the nodes to the placement aggregate and create instances
- change the hardware type and the interfaces back to the real ones

Make sure you change the drivers to the fake ones **before** providing the nodes as cleaning will otherwise wipe your production servers!

The reason to make all nodes available and manage access via the aggregate is that this is much faster than providing nodes one by one and relying on the resource tracker to find them. Enabling them one by one is required to make sure the instance name and the (pre-adoption) name of the server match.

The above recipe does not cover Neutron which, depending on your deployment, may need to be handled in addition.

4.2.6 Node retirement

Overview

Retiring nodes is a natural part of a servers life cycle, for instance when the end of the warranty is reached and the physical space is needed for new deliveries to install replacement capacity.

However, depending on the type of the deployment, removing nodes from service can be a full workflow by itself as it may include steps like moving applications to other hosts, cleaning sensitive data from disks or the BMC, or tracking the dismantling of servers from their racks.

Ironic provides some means to support such workflows by allowing to tag nodes as `retired` which will prevent any further scheduling of instances, but will still allow for other operations, such as cleaning, to happen (this marks an important difference to nodes which have the `maintenance` flag set).

How to use

When it is known that a node shall be retired, set the `retired` flag on the node with:

```
baremetal node set --retired node-001
```

This can be done irrespective of the state the node is in, so in particular while the node is `active`.

Note: An exception are nodes which are in `available`. For backwards compatibility reasons, these nodes need to be moved to `manageable` first. Trying to set the `retired` flag for `available` nodes will result in an error.

Optionally, a reason can be specified when a node is retired, e.g.:

```
baremetal node set --retired node-001 \
  --retired-reason "End of warranty for delivery abc123"
```

Upon instance deletion, an `active` node with the `retired` flag set will not move to `available`, but to `manageable`. The node will hence not be eligible for scheduling of new instances.

Equally, nodes with `retired` set to True cannot move from `manageable` to `available`: the `provide` verb is blocked. This is to prevent accidental re-use of nodes tagged for removal from the fleet. In order to move these nodes to `available` none the less, the `retired` field needs to be removed first. This can be done via:

```
baremetal node unset --retired node-001
```

In order to facilitate the identification of nodes marked for retirement, e.g. by other teams, ironic also allows to list all nodes which have the `retired` flag set:

```
baremetal node list --retired
```

4.2.7 RAID Configuration

Overview

Ironic supports RAID configuration for bare metal nodes. It allows operators to specify the desired RAID configuration via the OpenStackClient CLI or REST API. The desired RAID configuration is applied on the bare metal during manual cleaning.

The examples described here use the OpenStackClient CLI; please see the REST API reference for their corresponding REST API requests.

Prerequisites

The bare metal node needs to use a hardware type that supports RAID configuration. RAID interfaces may implement RAID configuration either in-band or out-of-band. Software RAID is supported on all hardware, although with some caveats - see *Software RAID* for details.

In-band RAID configuration (including software RAID) is done using the Ironic Python Agent ramdisk. For in-band hardware RAID configuration, a hardware manager which supports RAID should be bundled with the ramdisk.

Whether a node supports RAID configuration could be found using the CLI command `baremetal node validate <node>`. In-band RAID is usually implemented by the `agent` RAID interface.

Build agent ramdisk which supports RAID configuration

For doing in-band hardware RAID configuration, Ironic needs an agent ramdisk bundled with a hardware manager which supports RAID configuration for your hardware. For example, the *DIB support for Proliant Hardware Manager* should be used for HPE Proliant Servers.

Note: For in-band software RAID, the agent ramdisk does not need to be bundled with a hardware manager as the generic hardware manager in the Ironic Python Agent already provides (basic) support for software RAID.

RAID configuration JSON format

The desired RAID configuration and current RAID configuration are represented in JSON format.

Target RAID configuration

This is the desired RAID configuration on the bare metal node. Using the OpenStackClient CLI (or REST API), the operator sets `target_raid_config` field of the node. The target RAID configuration will be applied during manual cleaning.

Target RAID configuration is a dictionary having `logical_disks` as the key. The value for the `logical_disks` is a list of JSON dictionaries. It looks like:

```
{
  "logical_disks": [
    {<desired properties of logical disk 1>},
    {<desired properties of logical disk 2>},
    ...
  ]
}
```

If the `target_raid_config` is an empty dictionary, it unsets the value of `target_raid_config` if the value was set with previous RAID configuration done on the node.

Each dictionary of logical disk contains the desired properties of logical disk supported by the hardware type. These properties are discoverable by:

```
baremetal driver raid property list <driver name>
```

Mandatory properties

These properties must be specified for each logical disk and have no default values:

- `size_gb` - Size (Integer) of the logical disk to be created in GiB. `MAX` may be specified if the logical disk should use all of the remaining space available. This can be used only when backing physical disks are specified (see below).
- `raid_level` - RAID level for the logical disk. Ironic supports the following RAID levels: 0, 1, 2, 5, 6, 1+0, 5+0, 6+0.

Optional properties

These properties have default values and they may be overridden in the specification of any logical disk. None of these options are supported for software RAID.

- `volume_name` - Name of the volume. Should be unique within the Node. If not specified, volume name will be auto-generated.
- `is_root_volume` - Set to `true` if this is the root volume. At most one logical disk can have this set to `true`; the other logical disks must have this set to `false`. The `root device hint` will be saved, if the RAID interface is capable of retrieving it. This is `false` by default.

Backing physical disk hints

These hints are specified for each logical disk to let Ironic find the desired disks for RAID configuration. This is machine-independent information. This serves the use-case where the operator doesnt want to provide individual details for each bare metal node. None of these options are supported for software RAID.

- `share_physical_disks` - Set to `true` if this logical disk can share physical disks with other logical disks. The default value is `false`, except for software RAID which always shares disks.
- `disk_type` - `hdd` or `ssd`. If this is not specified, disk type will not be a criterion to find backing physical disks.
- `interface_type` - `sata` or `scsi` or `sas`. If this is not specified, interface type will not be a criterion to find backing physical disks.
- `number_of_physical_disks` - Integer, number of disks to use for the logical disk. Defaults to minimum number of disks required for the particular RAID level, except for software RAID which always spans all disks.

Backing physical disks

These are the actual machine-dependent information. This is suitable for environments where the operator wants to automate the selection of physical disks with a 3rd-party tool based on a wider range of attributes (eg. S.M.A.R.T. status, physical location). The values for these properties are hardware dependent.

- `controller` - The name of the controller as read by the RAID interface. In order to trigger the setup of a Software RAID via the Ironic Python Agent, the value of this property needs to be set to `software`.
- `physical_disks` - A list of physical disks to use as read by the RAID interface.

  For software RAID `physical_disks` is a list of device hints in the same format as used for *Specifying the disk for deployment (root device hints)*. The number of provided hints must match the expected number of backing devices (repeat the same hint if necessary).

Note: If properties from both Backing physical disk hints or Backing physical disks are specified, they should be consistent with each other. If they are not consistent, then the RAID configuration will fail (because the appropriate backing physical disks could not be found).

Examples for `target_raid_config`

Example 1. Single RAID disk of RAID level 5 with all of the space available. Make this the root volume to which Ironic deploys the image:

```
{
  "logical_disks": [
    {
      "size_gb": "MAX",
      "raid_level": "5",
      "is_root_volume": true
    }
  ]
}
```

Example 2. Two RAID disks. One with RAID level 5 of 100 GiB and make it root volume and use SSD. Another with RAID level 1 of 500 GiB and use HDD:

```
{
  "logical_disks": [
    {
      "size_gb": 100,
      "raid_level": "5",
      "is_root_volume": true,
      "disk_type": "ssd"
    },
    {
      "size_gb": 500,
      "raid_level": "1",
```

(continues on next page)

(continued from previous page)

```
      "disk_type": "hdd"
    }
  ]
}
```

Example 3. Single RAID disk. I know which disks and controller to use:

```
{
  "logical_disks": [
    {
      "size_gb": 100,
      "raid_level": "5",
      "controller": "Smart Array P822 in Slot 3",
      "physical_disks": ["6I:1:5", "6I:1:6", "6I:1:7"],
      "is_root_volume": true
    }
  ]
}
```

Example 4. Using backing physical disks:

```
{
  "logical_disks": [
    {
      "size_gb": 50,
      "raid_level": "1+0",
      "controller": "RAID.Integrated.1-1",
      "volume_name": "root_volume",
      "is_root_volume": true,
      "physical_disks": [
        "Disk.Bay.0:Encl.Int.0-1:RAID.Integrated.1-1",
        "Disk.Bay.1:Encl.Int.0-1:RAID.Integrated.1-1"
      ]
    },
    {
      "size_gb": 100,
      "raid_level": "5",
      "controller": "RAID.Integrated.1-1",
      "volume_name": "data_volume",
      "physical_disks": [
        "Disk.Bay.2:Encl.Int.0-1:RAID.Integrated.1-1",
        "Disk.Bay.3:Encl.Int.0-1:RAID.Integrated.1-1",
        "Disk.Bay.4:Encl.Int.0-1:RAID.Integrated.1-1"
      ]
    }
  ]
}
```

Example 5. Software RAID with two RAID devices:

```
{
  "logical_disks": [
    {
      "size_gb": 100,
      "raid_level": "1",
      "controller": "software"
    },
    {
      "size_gb": "MAX",
      "raid_level": "0",
      "controller": "software"
    }
  ]
}
```

Example 6. Software RAID, limiting backing block devices to exactly two devices with the size exceeding 100 GiB:

```
{
  "logical_disks": [
    {
      "size_gb": "MAX",
      "raid_level": "0",
      "controller": "software",
      "physical_disks": [
        {"size": "> 100"},
        {"size": "> 100"}
      ]
    }
  ]
}
```

Current RAID configuration

After target RAID configuration is applied on the bare metal node, Ironic populates the current RAID configuration. This is populated in the `raid_config` field in the Ironic node. This contains the details about every logical disk after they were created on the bare metal node. It contains details like RAID controller used, the backing physical disks used, WWN of each logical disk, etc. It also contains information about each physical disk found on the bare metal node.

To get the current RAID configuration:

```
baremetal node show <node-uuid-or-name>
```

Workflow

- Operator configures the bare metal node with a hardware type that has a `RAIDInterface` other than `no-raid`. For instance, for Software RAID, this would be `agent`.
- For in-band RAID configuration, operator builds an agent ramdisk which supports RAID configuration by bundling the hardware manager with the ramdisk. See *Build agent ramdisk which supports RAID configuration* for more information.
- Operator prepares the desired target RAID configuration as mentioned in *Target RAID configuration*. The target RAID configuration is set on the Ironic node:

```
baremetal node set <node-uuid-or-name> \
   --target-raid-config <JSON file containing target RAID configuration>
```

  The CLI command can accept the input from standard input also:

```
baremetal node set <node-uuid-or-name> \
   --target-raid-config -
```

- Create a JSON file with the RAID clean steps for manual cleaning. Add other clean steps as desired:

```
[{
  "interface": "raid",
  "step": "delete_configuration"
},
{
  "interface": "raid",
  "step": "create_configuration"
}]
```

Note: create_configuration doesnt remove existing disks. It is recommended to add delete_configuration before create_configuration to make sure that only the desired logical disks exist in the system after manual cleaning.

- Bring the node to `manageable` state and do a `clean` action to start cleaning on the node:

```
baremetal node clean <node-uuid-or-name> \
   --clean-steps <JSON file containing clean steps created above>
```

- After manual cleaning is complete, the current RAID configuration is reported in the `raid_config` field when running:

```
baremetal node show <node-uuid-or-name>
```

Software RAID

Building Linux software RAID in-band (via the Ironic Python Agent ramdisk) is supported starting with the Train release. It is requested by using the `agent` RAID interface and RAID configuration with all controllers set to `software`. You can find a software RAID configuration example in *Examples for target_raid_config*.

There are certain limitations to be aware of:

- Only the mandatory properties (plus the required `controller` property) from *Target RAID configuration* are currently supported.
- The number of created Software RAID devices must be 1 or 2. If there is only one Software RAID device, it has to be a RAID-1. If there are two, the first one has to be a RAID-1, while the RAID level for the second one can be 0, 1, 1+0, 5, or 6. As the first RAID device will be the deployment device, enforcing a RAID-1 reduces the risk of ending up with a non-booting node in case of a disk failure.
- Building RAID will fail if the target disks are already partitioned. Wipe the disks using e.g. the `erase_devices_metadata` clean step before building RAID:

```
[{
  "interface": "raid",
  "step": "delete_configuration"
},
{
  "interface": "deploy",
  "step": "erase_devices_metadata"
},
{
  "interface": "raid",
  "step": "create_configuration"
}]
```

- The final instance image must have the `mdadm` utility installed and needs to be able to detect software RAID devices at boot time (which is usually done by having the RAID drivers embedded in the images initrd).
- Regular cleaning will not remove RAID configuration (similarly to hardware RAID). To destroy RAID run the `delete_configuration` manual clean step.
- There is no support for partition images, only whole-disk images are supported with Software RAID. See *Add images to the Image service*. This includes flavors requesting dynamic creation of swap filesystems. Swap should be pre-allocated inside of a disk image partition layout.
- Images utilizing LVM for their root filesystem are not supported. Patches are welcome to explicitly support such functionality.
- If the root filesystem UUID is not known to Ironic via metadata, then the disk image layout **MUST** have the first partition consist of the root filesystem. Ironic is agnostic if the partition table is a DOS MBR or a GPT partition.

  Starting in Ironic 14.0.0 (Ussuri), the root filesystem UUID can be set and passed through to Ironic through the Glance Image Service `properties` sub-field `rootfs_uuid` for the image to be deployed.

Starting in Ironic 16.1.0 (Wallaby), similar functionality is available via the baremetal node `instance_info` field value `image_rootfs_uuid`. See *Using Bare Metal service as a standalone service* for more details on standalone usage including an example command.

- In UEFI mode, the Ironic Python Agent creates EFI system partitions (ESPs) for the bootloader and the boot configuration (grub.cfg or grubenv) on all holder devices. The content of these partitions is populated upon deployment from the deployed user image. Depending on how the partitions are mounted, the content of the partitions may get out of sync, e.g. when new kernels are installed or the bootloader is updated, so measures to keep these partitions in sync need to be taken. Note that starting with the Victoria release, the Ironic Python Agent configures a RAID-1 mirror for the ESPs, so no additional measures to ensure consistency of the ESPs should be required any longer.
- In BIOS mode, the Ironic Python Agent installs the boot loader onto all disks. While nothing is required for kernel or grub package updates, re-installing the bootloader on one disk, e.g. during a disk replacement, may require to re-install the bootloader on all disks. Otherwise, there is a risk of an incompatibility of the grub components stored on the device (i.e. stage1/boot.img in the MBR and stage1.5/core.img in the MBR gap) with the ones stored in /boot (stage2). This incompatibility can render the node unbootable if the wrong disk is selected for booting.
- Linux kernel device naming is not consistent across reboots for RAID devices and may be numbered in a distribution specific pattern. Operators will need to be mindful of this if a root device hint is utilized. A particular example of this is that the first md0 device on a Ubuntu based ramdisk may start as device md0, whereas on a Centos or Red Hat Enterprise Linux based ramdisk may start at device md127. After a reboot, these device names may change entirely.

Note: *Root device hints* should not be explicitly required to utilize software RAID. Candidate devices are chosen by sorting the usable device list looking for the smallest usable device which is then sorted by name. The secondary sort by name improves the odds for matching the first initialized block device. In the case of software RAID, they are always a little smaller than the primary block devices due to metadata overhead, which helps make them the most likely candidate devices.

Image requirements

Since Ironic needs to perform additional steps when deploying nodes with software RAID, there are some requirements the deployed images need to fulfill. Up to and including the Train release, the image needs to have its root file system on the first partition. Starting with Ussuri, the image can also have additional metadata to point Ironic to the partition with the root file system: for this, the image needs to set the `rootfs_uuid` property with the file system UUID of the root file system. One way to extract this UUID from an existing image is to download the image, mount it as a loopback device, and use `blkid`:

```
$ sudo losetup -f
$ sudo losetup /dev/loop0 /tmp/myimage.raw
$ sudo kpartx -a /dev/loop0
$ blkid
```

The pre-Ussuri approach, i.e. to have the root file system on the first partition, is kept as a fallback and hence allows software RAID deployments where Ironic does not have access to any image metadata (e.g. Ironic stand-alone).

Using RAID in nova flavor for scheduling

The operator can specify the *raid_level* capability in nova flavor for node to be selected for scheduling:

```
openstack flavor set my-baremetal-flavor --property capabilities:raid_level=
↪"1+0"
```

Developer documentation

In-band RAID configuration is done using IPA ramdisk. IPA ramdisk has support for pluggable hardware managers which can be used to extend the functionality offered by IPA ramdisk using stevedore plugins. For more information, see Ironic Python Agent Hardware Manager documentation.

The hardware manager that supports RAID configuration should do the following:

1. Implement a method named `create_configuration`. This method creates the RAID configuration as given in `target_raid_config`. After successful RAID configuration, it returns the current RAID configuration information which ironic uses to set `node.raid_config`.
2. Implement a method named `delete_configuration`. This method deletes all the RAID disks on the bare metal.
3. Return these two clean steps in `get_clean_steps` method with priority as 0. Example:

```
return [{'step': 'create_configuration',
         'interface': 'raid',
         'priority': 0},
        {'step': 'delete_configuration',
         'interface': 'raid',
         'priority': 0}]
```

4.2.8 BIOS Configuration

Overview

The Bare Metal service supports BIOS configuration for bare metal nodes. It allows administrators to retrieve and apply the desired BIOS settings via CLI or REST API. The desired BIOS settings are applied during manual cleaning.

Prerequisites

Bare metal servers must be configured by the administrator to be managed via ironic hardware type that supports BIOS configuration.

Enabling hardware types

Enable a specific hardware type that supports BIOS configuration. Refer to *Enabling drivers and hardware types* for how to enable a hardware type.

Enabling hardware interface

To enable the bios interface:

```
[DEFAULT]
enabled_bios_interfaces = no-bios
```

Append the actual bios interface name supported by the enabled hardware type to `enabled_bios_interfaces` with comma separated values in `ironic.conf`.

All available in-tree bios interfaces are listed in setup.cfg file in the source code tree, for example:

```
ironic.hardware.interfaces.bios =
    fake = ironic.drivers.modules.fake:FakeBIOS
    no-bios = ironic.drivers.modules.noop:NoBIOS
```

Retrieve BIOS settings

To retrieve the cached BIOS configuration from a specified node:

```
$ baremetal node bios setting list <node>
```

BIOS settings are cached on each node cleaning operation or when settings have been applied successfully via BIOS cleaning steps. The return of above command is a table of last cached BIOS settings from specified node. If `-f json` is added as suffix to above command, it returns BIOS settings as following:

```
[
  {
    "setting name":
      {
        "name": "setting name",
        "value": "value"
      }
  },
  {
    "setting name":
      {
        "name": "setting name",
        "value": "value"
      }
  },
  ...
]
```

To get a specified BIOS setting for a node:

```
$ baremetal node bios setting show <node> <setting-name>
```

If `-f json` is added as suffix to above command, it returns BIOS settings as following:

```
{
  "setting name":
    {
      "name": "setting name",
      "value": "value"
    }
}
```

Configure BIOS settings

Two *Manual cleaning* steps are available for managing nodes BIOS settings:

Factory reset

This cleaning step resets all BIOS settings to factory default for a given node:

```
{
  "target":"clean",
  "clean_steps": [
    {
      "interface": "bios",
      "step": "factory_reset"
    }
  ]
}
```

The `factory_reset` cleaning step does not require any arguments, as it resets all BIOS settings to factory defaults.

Apply BIOS configuration

This cleaning step applies a set of BIOS settings for a node:

```
{
  "target":"clean",
  "clean_steps": [
    {
      "interface": "bios",
      "step": "apply_configuration",
      "args": {
        "settings": [
          {
            "name": "name",
            "value": "value"
```

(continues on next page)

(continued from previous page)

```
            },
            {
              "name": "name",
              "value": "value"
            }
          ]
        }
      }
    ]
}
```

The representation of `apply_configuration` cleaning step follows the same format of *Manual cleaning*. The desired BIOS settings can be provided via the `settings` argument which contains a list of BIOS options to be applied, each BIOS option is a dictionary with `name` and `value` keys.

To check whether the desired BIOS configuration is set properly, use the command mentioned in the *Retrieve BIOS settings* section.

Note: When applying BIOS settings to a node, vendor-specific driver may take the given BIOS settings from the argument and compare them with the current BIOS settings on the node and only apply when there is a difference.

4.2.9 Rescue Mode

Overview

The Bare Metal Service supports putting nodes in rescue mode using hardware types that support rescue interfaces. The hardware types utilizing ironic-python-agent with `PXE/Virtual Media` based boot interface can support rescue operation when configured appropriately.

Note: The rescue operation is currently supported only when tenant networks use DHCP to obtain IP addresses.

Rescue operation can be used to boot nodes into a rescue ramdisk so that the `rescue` user can access the node, in order to provide the ability to access the node in case access to OS is not possible. For example, if there is a need to perform manual password reset or data recovery in the event of some failure, rescue operation can be used.

Configuring The Bare Metal Service

Configure the Bare Metal Service appropriately so that the service has the information needed to boot the ramdisk before a user tries to initiate rescue operation. This will differ somewhat between different deploy environments, but an example of how to do this is outlined below:

1. Create and configure ramdisk that supports rescue operation. Please see *Building or downloading a deploy ramdisk image* for detailed instructions to build a ramdisk.
2. Configure a network to use for booting nodes into the rescue ramdisk in neutron, and note the UUID or name of this network. This is required if youre using the neutron DHCP provider and have Bare Metal Service managing ramdisk booting (the default). This can be the same network as your cleaning or tenant network (for flat network). For an example of how to configure new networks with Bare Metal Service, see the *Configure the Networking service for bare metal provisioning* documentation.
3. Add the unique name or UUID of your rescue network to `ironic.conf`:

   ```
   [neutron]
   rescuing_network=<RESCUE_UUID_OR_NAME>
   ```

   Note: This can be set per node via driver_info[rescuing_network]

4. Restart the ironic conductor service.
5. Specify a rescue kernel and ramdisk or rescue ISO compatible with the nodes driver for pxe based boot interface or virtual-media based boot interface respectively.

   Example for pxe based boot interface:

   ```
   baremetal node set $NODE_UUID \
       --driver-info rescue_ramdisk=$RESCUE_INITRD_UUID \
       --driver-info rescue_kernel=$RESCUE_VMLINUZ_UUID
   ```

   See *Add images to the Image service* for details. If you are not using Image service, it is possible to provide images to Bare Metal service via hrefs.

After this, The Bare Metal Service should be ready for `rescue` operation. Test it out by attempting to rescue an active node and connect to the instance using ssh, as given below:

```
baremetal node rescue $NODE_UUID \
    --rescue-password <PASSWORD> --wait

ssh rescue@$INSTANCE_IP_ADDRESS
```

To move a node back to active state after using rescue mode you can use `unrescue`. Please unmount any filesystems that were manually mounted before proceeding with unrescue. The node unrescue can be done as given below:

```
baremetal node unrescue $NODE_UUID
```

`rescue` and `unrescue` operations can also be triggered via the Compute Service using the following commands:

```
openstack server rescue --password <password> <server>

openstack server unrescue <server>
```

4.2.10 Boot From Volume

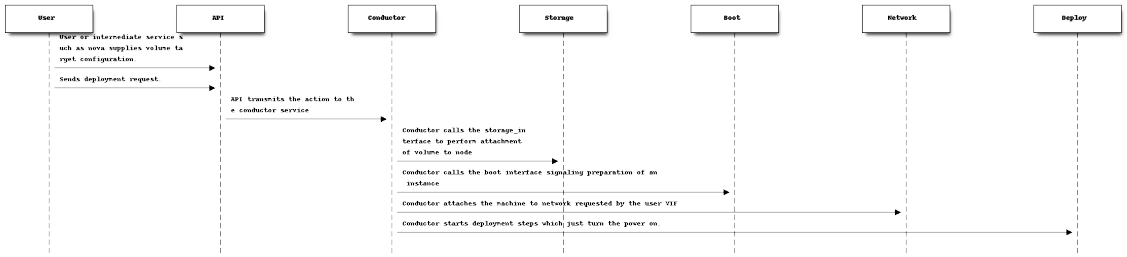
Overview

The Bare Metal service supports booting from a Cinder iSCSI volume as of the Pike release. This guide will primarily deal with this use case, but will be updated as more paths for booting from a volume, such as FCoE, are introduced.

The boot from volume is supported on both legacy BIOS and UEFI (iPXE binary for EFI booting) boot mode. We need to perform with suitable images which will be created by diskimage-builder tool.

How this works - From Ironics point of view

In essence, ironic sets the stage for the process, by providing the required information to the boot interface to facilitate the configuration of the the node OR the iPXE boot templates such that the node CAN be booted.



In this example, the boot interface does the heavy lifting. For drivers the `irmc` and `ilo` hardware types with hardware type specific boot interfaces, they are able to signal via an out of band mechanism to the baremetal nodes BMC that the integrated iSCSI initiators are to connect to the supplied volume target information.

In most hardware this would be the network cards of the machine.

In the case of the `ipxe` boot interface, templates are created on disk which point to the iscsi target information that was either submitted as part of the volume target, or when integrated with Nova, what was requested as the baremetals boot from volume disk upon requesting the instance.

In terms of network access, both interface methods require connectivity to the iscsi target. In the vendor driver specific path, additional network configuration options may be available to allow separation of standard network traffic and instance network traffic. In the iPXE case, this is not possible as the OS userspace re-configures the iSCSI connection after detection inside the OS ramdisk boot.

An iPXE user *may* be able to leverage multiple VIFs, one specifically set to be set with `pxe_enabled` to handle the initial instance boot and back-end storage traffic where as external facing network traffic occurs on a different interface. This is a common pattern in iSCSI based deployments in the physical realm.

Prerequisites

Currently booting from a volume requires:

- Bare Metal service version 9.0.0
- Bare Metal API microversion 1.33 or later
- A driver that utilizes the *PXE boot mechanism*. Currently booting from a volume is supported by the reference drivers that utilize PXE boot mechanisms when iPXE is enabled.
- iPXE is an explicit requirement, as it provides the mechanism that attaches and initiates booting from an iSCSI volume.
- Metadata services need to be configured and available for the instance images to obtain configuration such as keys. Configuration drives are not supported due to minimum disk extension sizes.

Conductor Configuration

In ironic.conf, you can specify a list of enabled storage interfaces. Check `[DEFAULT]enabled_storage_interfaces` in your ironic.conf to ensure that your desired interface is enabled. For example, to enable the `cinder` and `noop` storage interfaces:

```
[DEFAULT]
enabled_storage_interfaces = cinder,noop
```

If you want to specify a default storage interface rather than setting the storage interface on a per node basis, set `[DEFAULT]default_storage_interface` in ironic.conf. The `default_storage_interface` will be used for any node that doesnt have a storage interface defined.

Node Configuration

Storage Interface

You will need to specify what storage interface the node will use to handle storage operations. For example, to set the storage interface to `cinder` on an existing node:

```
baremetal node set --storage-interface cinder $NODE_UUID
```

A default storage interface can be specified in ironic.conf. See the *Conductor Configuration* section for details.

iSCSI Configuration

In order for a bare metal node to boot from an iSCSI volume, the `iscsi_boot` capability for the node must be set to `True`. For example, if you want to update an existing node to boot from volume:

```
baremetal node set --property capabilities=iscsi_boot:True $NODE_UUID
```

You will also need to create a volume connector for the node, so the storage interface will know how to communicate with the node for storage operation. In the case of iSCSI, you will need to provide an iSCSI Qualifying Name (IQN) that is unique to your SAN. For example, to create a volume connector for iSCSI:

```
baremetal volume connector create \
         --node $NODE_UUID --type iqn --connector-id iqn.2017-08.org.
↪openstack.$NODE_UUID
```

Image Creation

We use `disk-image-create` in diskimage-builder tool to create images for boot from volume feature. Some required elements for this mechanism for corresponding boot modes are as following:

- Legacy BIOS boot mode: `iscsi-boot` element.
- UEFI boot mode: `iscsi-boot` and `block-device-efi` elements.

An example below:

```
export IMAGE_NAME=<image_name>
export DIB_CLOUD_INIT_DATASOURCES="ConfigDrive, OpenStack"
disk-image-create centos7 vm cloud-init-datasources dhcp-all-interfaces iscsi-
↪boot dracut-regenerate block-device-efi -o $IMAGE_NAME
```

Note:

- For CentOS images, we must add dependent element named `dracut-regenerate` during image creation. Otherwise, the image creation will fail with an error.
- For Ubuntu images, we only support `iscsi-boot` element without `dracut-regenerate` element during image creation.

Advanced Topics

Use without the Compute Service

As discussed in other sections, the Bare Metal service has a concept of a *connector* that is used to represent an interface that is intended to be utilized to attach the remote volume.

In addition to the connectors, we have a concept of a *target* that can be defined via the API. While a user of this feature through the Compute service would automatically have a new target record created for them, it is not explicitly required, and can be performed manually.

A target record can be created using a command similar to the example below:

```
baremetal volume target create \
          --node $NODE_UUID --type iscsi --boot-index 0 --volume $VOLUME_UUID
```

Note: A `boot-index` value of `0` represents the boot volume for a node. As the `boot-index` is per-node in sequential order, only one boot volume is permitted for each node.

Use Without Cinder

In the Rocky release, an `external` storage interface is available that can be utilized without a Block Storage Service installation.

Under normal circumstances the `cinder` storage interface interacts with the Block Storage Service to orchestrate and manage attachment and detachment of volumes from the underlying block service system.

The `external` storage interface contains the logic to allow the Bare Metal service to determine if the Bare Metal node has been requested with a remote storage volume for booting. This is in contrast to the default `noop` storage interface which does not contain logic to determine if the node should or could boot from a remote volume.

It must be noted that minimal configuration or value validation occurs with the `external` storage interface. The `cinder` storage interface contains more extensive validation, that is likely un-necessary in a `external` scenario.

Setting the external storage interface:

```
baremetal node set --storage-interface external $NODE_UUID
```

Setting a volume:

```
baremetal volume target create --node $NODE_UUID \
    --type iscsi --boot-index 0 --volume-id $VOLUME_UUID \
    --property target_iqn="iqn.2010-10.com.example:vol-X" \
    --property target_lun="0" \
    --property target_portal="192.168.0.123:3260" \
    --property auth_method="CHAP" \
    --property auth_username="ABC" \
    --property auth_password="XYZ" \
```

Ensure that no image_source is defined:

```
baremetal node unset \
    --instance-info image_source $NODE_UUID
```

Deploy the node:

```
baremetal node deploy $NODE_UUID
```

Upon deploy, the boot interface for the baremetal node will attempt to either create iPXE configuration OR set boot parameters out-of-band via the management controller. Such action is boot interface specific and may not support all forms of volume target configuration. As of the Rocky release, the bare metal

service does not support writing an Operating System image to a remote boot from volume target, so that also must be ensured by the user in advance.

Records of volume targets are removed upon the node being undeployed, and as such are not persistent across deployments.

Cinder Multi-attach

Volume multi-attach is a function that is commonly performed in computing clusters where dedicated storage subsystems are utilized. For some time now, the Block Storage service has supported the concept of multi-attach. However, the Compute service, as of the Pike release, does not yet have support to leverage multi-attach. Concurrently, multi-attach requires the backend volume driver running as part of the Block Storage service to contain support for multi-attach volumes.

When support for storage interfaces was added to the Bare Metal service, specifically for the `cinder` storage interface, the concept of volume multi-attach was accounted for, however has not been fully tested, and is unlikely to be fully tested until there is Compute service integration as well as volume driver support.

The data model for storage of volume targets in the Bare Metal service has no constraints on the same target volume from being utilized. When interacting with the Block Storage service, the Bare Metal service will prevent the use of volumes that are being reported as `in-use` if they do not explicitly support multi-attach.

4.2.11 Multi-tenancy in the Bare Metal service

Overview

It is possible to use dedicated tenant networks for provisioned nodes, which extends the current Bare Metal service capabilities of providing flat networks. This works in conjunction with the Networking service to allow provisioning of nodes in a separate provisioning network. The result of this is that multiple tenants can use nodes in an isolated fashion. However, this configuration does not support trunk ports belonging to multiple networks.

Concepts

Network interfaces

Network interface is one of the driver interfaces that manages network switching for nodes. There are 3 network interfaces available in the Bare Metal service:

- `noop` interface is used for standalone deployments, and does not perform any network switching;
- `flat` interface places all nodes into a single provider network that is pre-configured on the Networking service and physical equipment. Nodes remain physically connected to this network during their entire life cycle.
- `neutron` interface provides tenant-defined networking through the Networking service, separating tenant networks from each other and from the provisioning and cleaning provider networks. Nodes will move between these networks during their life cycle. This interface requires Networking service support for the switches attached to the baremetal servers so they can be programmed.

Local link connection

The Bare Metal service allows `local_link_connection` information to be associated with Bare Metal ports. This information is provided to the Networking services ML2 driver when a Virtual Interface (VIF) is attached. The ML2 driver uses the information to plug the specified port to the tenant network.

Table 25: `local_link_connection` fields

Field	Description
`switch_id`	Required. Identifies a switch and can be a MAC address or an OpenFlow-based `datapath_id`.
`port_id`	Required. Port ID on the switch/Smart NIC, for example, Gig0/1, rep0-0.
`switch_info`	Optional. Used to distinguish different switch models or other vendor-specific identifier. Some ML2 plugins may require this field.
`hostname`	Required in case of a Smart NIC port. Hostname of Smart NIC device.

Note: This isnt applicable to Infiniband ports because the network topology is discoverable by the Infiniband Subnet Manager. If specified, local_link_connection information will be ignored. If port is Smart NIC port then:

1. `port_id` is the representor port name on the Smart NIC.
2. `switch_id` is not mandatory.

Physical networks

A Bare Metal port may be associated with a physical network using its `physical_network` field. The Bare Metal service uses this information when mapping between virtual ports in the Networking service and physical ports and port groups in the Bare Metal service. A ports physical network field is optional, and if not set then any virtual port may be mapped to that port, provided that no free Bare Metal port with a suitable physical network assignment exists.

The physical network of a port group is defined by the physical network of its constituent ports. The Bare Metal service ensures that all ports in a port group have the same value in their physical network field.

When attaching a virtual interface (VIF) to a node, the following ordered criteria are used to select a suitable unattached port or port group:

- Require ports or port groups to not have a physical network or to have a physical network that matches one of the VIFs allowed physical networks.
- Prefer ports and port groups that have a physical network to ports and port groups that do not have a physical network.
- Prefer port groups to ports. Prefer ports with PXE enabled.

Configuring the Bare Metal service

See the *Configure tenant networks* section in the installation guide for the Bare Metal service.

Configuring nodes

1. Ensure that your python-ironicclient version and requested API version are sufficient for your requirements.

   - Multi-tenancy support was added in API version 1.20, and is supported by python-ironicclient version 1.5.0 or higher.
   - Physical network support for ironic ports was added in API version 1.34, and is supported by python-ironicclient version 1.15.0 or higher.
   - Smart NIC support for ironic ports was added in API version 1.53, and is supported by python-ironicclient version 2.7.0 or higher.

   The following examples assume you are using python-ironicclient version 2.7.0 or higher.

   Export the following variable:

   ```
   export OS_BAREMETAL_API_VERSION=<API version>
   ```

2. The nodes `network_interface` field should be set to a valid network interface. Valid interfaces are listed in the `[DEFAULT]/enabled_network_interfaces` configuration option in the ironic-conductors configuration file. Set it to `neutron` to use the Networking services ML2 driver:

   ```
   baremetal node create --network-interface neutron --driver ipmi
   ```

   Note: If the `[DEFAULT]/default_network_interface` configuration option is set, the `--network-interface` option does not need to be specified when creating the node.

3. To update an existing nodes network interface to `neutron`, use the following commands:

   ```
   baremetal node set $NODE_UUID_OR_NAME \
       --network-interface neutron
   ```

4. Create a port as follows:

   ```
   baremetal port create $HW_MAC_ADDRESS --node $NODE_UUID \
       --local-link-connection switch_id=$SWITCH_MAC_ADDRESS \
       --local-link-connection switch_info=$SWITCH_HOSTNAME \
       --local-link-connection port_id=$SWITCH_PORT \
       --pxe-enabled true \
       --physical-network physnet1
   ```

   An Infiniband port requires client ID, while local link connection information will be populated by Infiniband Subnet Manager. The client ID consists of <12-byte vendor prefix>:<8 byte port GUID>. There is no standard process for deriving the ports MAC address ($HW_MAC_ADDRESS); it is vendor specific. For example, Mellanox ConnectX Family Devices prefix is ff:00:00:00:00:00:02:00:00:02:c9:00. If port GUID was f4:52:14:03:00:38:39:81 the client ID would be ff:00:00:00:00:00:02:00:00:02:c9:00:f4:52:14:03:00:38:39:81. Mellanox

ConnectX Family Devices HW_MAC_ADDRESS consists of 6 bytes; the port GUIDs lower 3 and higher 3 bytes. In this example it would be f4:52:14:38:39:81. Putting it all together, create an Infiniband port as follows:

```
baremetal port create $HW_MAC_ADDRESS --node $NODE_UUID \
    --pxe-enabled true \
    --extra client-id=$CLIENT_ID \
    --physical-network physnet1
```

5. Create a Smart NIC port as follows:

```
baremetal port create $HW_MAC_ADDRESS --node $NODE_UUID \
    --local-link-connection hostname=$HOSTNAME \
    --local-link-connection port_id=$REP_NAME \
    --pxe-enabled true \
    --physical-network physnet1 \
    --is-smartnic
```

   A Smart NIC port requires `hostname` which is the hostname of the Smart NIC, and `port_id` which is the representor port name within the Smart NIC.

6. Check the port configuration:

```
baremetal port show $PORT_UUID
```

After these steps, the provisioning of the created node will happen in the provisioning network, and then the node will be moved to the tenant network that was requested.

Configuring the Networking service

In addition to configuring the Bare Metal service some additional configuration of the Networking service is required to ensure ports for bare metal servers are correctly programmed. This configuration will be determined by the Bare Metal service network interfaces you have enabled and which top of rack switches you have in your environment.

`flat` network interface

In order for Networking service ports to correctly operate with the Bare Metal service `flat` network interface the `baremetal` ML2 mechanism driver from networking-baremetal needs to be loaded into the Networking service configuration. This driver understands that the switch should be already configured by the admin, and will mark the networking service ports as successfully bound as nothing else needs to be done.

1. Install the `networking-baremetal` library

```
$ pip install networking-baremetal
```

2. Enable the `baremetal` driver in the Networking service ML2 configuration file

```
[ml2]
mechanism_drivers = ovs,baremetal
```

`neutron` network interface

The `neutron` network interface allows the Networking service to program the physical top of rack switches for the bare metal servers. To do this an ML2 mechanism driver which supports the `baremetal` VNIC type for the make and model of top of rack switch in the environment must be installed and enabled.

This is a list of known top of rack ML2 mechanism drivers which work with the `neutron` network interface:

Cisco Nexus 9000 series To install and configure this ML2 mechanism driver see Nexus Mechanism Driver Installation Guide.

FUJITSU CFX2000 `networking-fujitsu` ML2 driver supports this switch. The documentation is available here.

Networking Generic Switch This is an ML2 mechanism driver built for testing against virtual bare metal environments and some switches that are not covered by hardware specific ML2 mechanism drivers. More information is available in the projects README.

4.2.12 Port groups support

The Bare Metal service supports static configuration of port groups (bonds) in the instances via config-drive. See kernel documentation on bonding to see why it may be useful and how it is setup in linux. The sections below describe how to make use of them in the Bare Metal service.

Switch-side configuration

If port groups are desired in the ironic deployment, they need to be configured on the switches. It needs to be done manually, and the mode and properties configured on the switch have to correspond to the mode and properties that will be configured on the ironic side, as bonding mode and properties may be named differently on your switch, or have possible values different from the ones described in kernel documentation on bonding. Please refer to your switch configuration documentation for more details.

Provisioning and cleaning cannot make use of port groups if they need to boot the deployment ramdisk via (i)PXE. If your switches or desired port group configuration do not support port group fallback, which will allow port group members to be used by themselves, you need to set port groups `standalone_ports_supported` value to be `False` in ironic, as it is `True` by default.

Physical networks

If any port in a port group has a physical network, then all ports in that port group must have the same physical network.

In order to change the physical network of the ports in a port group, all ports must first be removed from the port group, before changing their physical networks (to the same value), then adding them back to the port group.

See *physical networks* for further information on using physical networks in the Bare Metal service.

Port groups configuration in the Bare Metal service

Port group configuration is supported in ironic API microversions 1.26, the CLI commands below specify it for completeness.

1. When creating a port group, the node to which it belongs must be specified, along with, optionally, its name, address, mode, properties, and if it supports fallback to standalone ports:

```
baremetal port group create \
--node $NODE_UUID --name test --address fa:ab:25:48:fd:ba --mode 802.3ad \
--property miimon=100 --property xmit_hash_policy="layer2+3" \
--support-standalone-ports
```

   A port group can also be updated with `baremetal port group set` command, see its help for more details.

   If an address is not specified, the port group address on the deployed instance will be the same as the address of the neutron port that is attached to the port group. If the neutron port is not attached, the port group will not be configured.

Note: In standalone mode, port groups have to be configured manually. It can be done either statically inside the image, or by generating the configdrive and adding it to the nodes `instance_info`. For more information on how to configure bonding via configdrive, refer to cloud-init documentation and code. cloud-init version 0.7.7 or later is required for bonding configuration to work.

The following is a simple sample for configuring bonding via configdrive:

When booting an instance, it needs to add user-data file for configuring bonding via `--user-data` option. For example:

```
{
  "networks": [
    {
      "type": "physical",
      "name": "eth0",
      "mac_address": "fa:ab:25:48:fd:ba"
    },
    {
      "type": "physical",
      "name": "eth1",
      "mac_address": "fa:ab:25:48:fd:ab"
    },
    {
      "type": "bond",
      "name": "bond0",
      "bond_interfaces": [
        "eth0", "eth1"
        ],
        "mode": "active-backup"
    }
  ]
}
```

If the port groups address is not explicitly set in standalone mode, it will be set automatically by the process described in kernel documentation on bonding.

During interface attachment, port groups have higher priority than ports, so they will be used first. (It is not yet possible to specify which one is desired, a port group or a port, in an interface attachment request). Port groups that dont have any ports will be ignored.

The mode and properties values are described in the kernel documentation on bonding. The default port group mode is `active-backup`, and this default can be changed by setting the `[DEFAULT]default_portgroup_mode` configuration option in the ironic API service configuration file.

2. Associate ports with the created port group.

   It can be done on port creation:

   ```
   baremetal port create \
   --node $NODE_UUID --address fa:ab:25:48:fd:ba --port-group test
   ```

   Or by updating an existing port:

   ```
   baremetal port set $PORT_UUID --port-group $PORT_GROUP_UUID
   ```

   When updating a port, the node associated with the port has to be in `enroll`, `manageable`, or `inspecting` states. A port group can have the same or different address as individual ports.

3. Boot an instance (or node directly, in case of using standalone ironic) providing an image that has cloud-init version 0.7.7 or later and supports bonding.

When the deployment is done, you can check that the port group is set up properly by running the following command in the instance:

```
cat /proc/net/bonding/bondX
```

where `X` is a number autogenerated by cloud-init for each configured port group, in no particular order. It starts with 0 and increments by 1 for every configured port group.

Link aggregation/teaming on windows

Portgroups are supported for Windows Server images, which can created by *Building images for Windows* instruction.

You can customise an instance after it is launched along with script file in `Configuration` of `Instance` and selected `Configuration Drive` option. Then ironic virt driver will generate network metadata and add all the additional information, such as bond mode, transmit hash policy, MII link monitoring interval, and of which links the bond consists. The information in InstanceMetadata will be used afterwards to generate the config drive.

4.2.13 Configuring Web or Serial Console

Overview

There are two types of console which are available in Bare Metal service, one is web console (*Node web console*) which is available directly from web browser, another is serial console (*Node serial console*).

Node web console

The web console can be configured in Bare Metal service in the following way:

- Install shellinabox in ironic conductor node. For RHEL/CentOS, shellinabox package is not present in base repositories, user must enable EPEL repository, you can find more from FedoraProject page.

  Note: shellinabox is no longer maintained by the authorized author. This is a fork of the project on GitHub that aims to continue with maintenance of the shellinabox project.

  Installation example:

  Ubuntu:

  ```
  sudo apt-get install shellinabox
  ```

  RHEL8/CentOS8/Fedora:

  ```
  sudo dnf install shellinabox
  ```

  You can find more about shellinabox on the shellinabox page.

  You can optionally use the SSL certificate in shellinabox. If you want to use the SSL certificate in shellinabox, you should install openssl and generate the SSL certificate.

  1. Install openssl, for example:

     Ubuntu:

     ```
     sudo apt-get install openssl
     ```

     RHEL8/CentOS8/Fedora:

     ```
     sudo dnf install openssl
     ```

  2. Generate the SSL certificate, here is an example, you can find more about openssl on the openssl page:

     ```
     cd /tmp/ca
     openssl genrsa -des3 -out my.key 1024
     openssl req -new -key my.key  -out my.csr
     cp my.key my.key.org
     openssl rsa -in my.key.org -out my.key
     openssl x509 -req -days 3650 -in my.csr -signkey my.key -out my.crt
     cat my.crt my.key > certificate.pem
     ```

- Customize the console section in the Bare Metal service configuration file (/etc/ironic/ironic.conf), if you want to use SSL certificate in shellinabox, you should specify `terminal_cert_dir`. for example:

```
[console]

#
# Options defined in ironic.drivers.modules.console_utils
#

# Path to serial console terminal program. Used only by Shell
# In A Box console. (string value)
#terminal=shellinaboxd

# Directory containing the terminal SSL cert (PEM) for serial
# console access. Used only by Shell In A Box console. (string
# value)
terminal_cert_dir=/tmp/ca

# Directory for holding terminal pid files. If not specified,
# the temporary directory will be used. (string value)
#terminal_pid_dir=<None>

# Time interval (in seconds) for checking the status of
# console subprocess. (integer value)
#subprocess_checking_interval=1

# Time (in seconds) to wait for the console subprocess to
# start. (integer value)
#subprocess_timeout=10
```

- Append console parameters for bare metal PXE boot in the Bare Metal service configuration file (/etc/ironic/ironic.conf). See the reference for configuration in *Appending kernel parameters to boot instances*.
- Enable the `ipmitool-shellinabox` console interface, for example:

```
[DEFAULT]
enabled_console_interfaces = ipmitool-shellinabox,no-console
```

- Configure node web console.

  If the node uses a hardware type, for example `ipmi`, set the nodes console interface to `ipmitool-shellinabox`:

```
baremetal node set <node> --console-interface ipmitool-shellinabox
```

  Enable the web console, for example:

```
baremetal node set <node> \
    --driver-info <terminal_port>=<customized_port>
baremetal node console enable <node>
```

Check whether the console is enabled, for example:

```
baremetal node validate <node>
```

Disable the web console, for example:

```
baremetal node console disable  <node>
baremetal node unset <node> --driver-info <terminal_port>
```

The <terminal_port> is driver dependent. The actual name of this field can be checked in driver properties, for example:

```
baremetal driver property list <driver>
```

For the ipmi hardware type, this option is ipmi_terminal_port. Give a customized port number to <customized_port>, for example 8023, this customized port is used in web console url.

Get web console information for a node as follows:

```
baremetal node console show <node>
+-----------------+--------------------------------------------------------
↪---------------+
| Property        | Value                                                  ⌋
↪               |
+-----------------+--------------------------------------------------------
↪---------------+
| console_enabled | True                                                   ⌋
↪               |
| console_info    | {u'url': u'http://<url>:<customized_port>', u'type': u
↪'shellinabox'} |
+-----------------+--------------------------------------------------------
↪---------------+
```

You can open web console using above url through web browser. If console_enabled is false, console_info is None, web console is disabled. If you want to launch web console, see the Configure node web console part.

Note: An error message you may encounter when enabling the console can read Console subprocess failed to start. Timeout or error while waiting for console subprocess to start for node along with [server] Failed to find any available port!. This error is coming from shellinabox itself, not from the communication with the BMC. One potential cause for this issue is that there are already shellinabox daemons running which block the configured port (remove them if appropriate and retry to enable the console).

Node serial console

Serial consoles for nodes are implemented using socat. It is supported by the `ipmi` and `irmc` hardware types.

Serial consoles can be configured in the Bare Metal service as follows:

- Install socat on the ironic conductor node. Also, `socat` needs to be in the $PATH environment variable that the ironic-conductor service uses.

  Installation example:

  Ubuntu:

  ```
  sudo apt-get install socat
  ```

  RHEL8/CentOS8/Fedora:

  ```
  sudo dnf install socat
  ```

- Append console parameters for bare metal PXE boot in the Bare Metal service configuration file. See the reference on how to configure them in *Appending kernel parameters to boot instances*.
- Enable the `ipmitool-socat` console interface, for example:

  ```
  [DEFAULT]
  enabled_console_interfaces = ipmitool-socat,no-console
  ```

- Configure node console.

  If the node uses a hardware type, for example `ipmi`, set the nodes console interface to `ipmitool-socat`:

  ```
  baremetal node set <node> --console-interface ipmitool-socat
  ```

  Enable the serial console, for example:

  ```
  baremetal node set <node> --driver-info ipmi_terminal_port=<port>
  baremetal node console enable <node>
  ```

  Check whether the serial console is enabled, for example:

  ```
  baremetal node validate <node>
  ```

  Disable the serial console, for example:

  ```
  baremetal node console disable  <node>
  baremetal node unset <node> --driver-info <ipmi_terminal_port>
  ```

Serial console information is available from the Bare Metal service. Get serial console information for a node from the Bare Metal service as follows:

```
baremetal node console show <node>
+----------------+--------------------------------------------------------------
↪-----------+
```

(continues on next page)

(continued from previous page)

```
| Property        | Value                                                          ␣
↪           |
+-----------------+---------------------------------------------------------------
↪-----------+
| console_enabled | True                                                           ␣
↪           |
| console_info    | {u'url': u'tcp://<host>:<port>', u'type': u'socat'}           ␣
↪           |
+-----------------+---------------------------------------------------------------
↪-----------+
```

If `console_enabled` is `false` or `console_info` is `None` then the serial console is disabled. If you want to launch serial console, see the `Configure node console`.

Node serial console of the Bare Metal service is compatible with the serial console of the Compute service. Hence, serial consoles to Bare Metal nodes can be seen and interacted with via the Dashboard service. In order to achieve that, you need to follow the documentation for Serial Console from the Compute service.

Configuring HA

When using Bare Metal serial console under High Availability (HA) configuration, you may consider some settings below.

- If you use HAProxy, you may need to set the timeout for both client and server sides with appropriate values. Here is an example of the configuration for the timeout parameter.

```
frontend nova_serial_console
  bind 192.168.20.30:6083
  timeout client 10m  # This parameter is necessary
  use_backend nova_serial_console if <...>

backend nova_serial_console
  balance source
  timeout server 10m  # This parameter is necessary
  option  tcpka
  option  tcplog
  server  controller01 192.168.30.11:6083 check inter 2000 rise 2 fall 5
  server  controller02 192.168.30.12:6083 check inter 2000 rise 2 fall 5
```

- The Compute services caching feature may need to be enabled in order to make the Bare Metal serial console work under a HA configuration. Here is an example of caching configuration in `nova.conf`.

```
[cache]
enabled = true
backend = dogpile.cache.memcached
memcache_servers = memcache01:11211,memcache02:11211,memcache03:11211
```

4.2.14 Notifications

Ironic, when configured to do so, will emit notifications over a message bus that indicate different events that occur within the service. These can be consumed by any external service. Examples may include a billing or usage system, a monitoring data store, or other OpenStack services. This page describes how to enable notifications and the different kinds of notifications that ironic may emit. The external consumer will see notifications emitted by ironic as JSON objects structured in the following manner:

```
{
    "priority": <string, defined by the sender>,
    "event_type": <string, defined by the sender>,
    "timestamp": <string, the isotime of when the notification emitted>,
    "publisher_id": <string, defined by the sender>,
    "message_id": <uuid, generated by oslo>,
    "payload": <json serialized dict, defined by the sender>
}
```

Configuration

To enable notifications with ironic, there are two configuration options in ironic.conf that must be adjusted.

The first option is the `notification_level` option in the `[DEFAULT]` section of the configuration file. This can be set to debug, info, warning, error, or critical, and determines the minimum priority level for which notifications are emitted. For example, if the option is set to warning, all notifications with priority level warning, error, or critical are emitted, but not notifications with priority level debug or info. For information about the semantics of each log level, see the OpenStack logging standards[1]. If this option is unset, no notifications will be emitted. The priority level of each available notification is documented below.

The second option is the `transport_url` option in the `[oslo_messaging_notifications]` section of the configuration. This determines the message bus used when sending notifications. If this is unset, the default transport used for RPC is used.

All notifications are emitted on the ironic_versioned_notifications topic in the message bus. Generally, each type of message that traverses the message bus is associated with a topic describing what the message is about. For more information, see the documentation of your chosen message bus, such as the RabbitMQ documentation[2].

Note that notifications may be lossy, and theres no guarantee that a notification will make it across the message bus to a consumer.

[1] https://wiki.openstack.org/wiki/LoggingStandards#Log_level_definitions
[2] https://www.rabbitmq.com/documentation.html

Versioning

Each notification has an associated version in the ironic_object.version field of the payload. Consumers are guaranteed that microversion bumps will add new fields, while macroversion bumps are backwards-incompatible and may have fields removed.

Versioned notifications are emitted by default to the *ironic_versioned_notifications* topic. This can be changed and it is configurable in the ironic.conf with the *versioned_notifications_topics* config option.

Available notifications

The notifications that ironic emits are described here. They are listed (alphabetically) by service first, then by event_type. All examples below show payloads before serialization to JSON.

ironic-api notifications

Resources CRUD notifications

These notifications are emitted from API service when ironic resources are modified as part of create, update, or delete (CRUD)[3] procedures. All CRUD notifications are emitted at INFO level, except for error status that is emitted at ERROR level.

List of CRUD notifications for chassis:

- `baremetal.chassis.create.start`
- `baremetal.chassis.create.end`
- `baremetal.chassis.create.error`
- `baremetal.chassis.update.start`
- `baremetal.chassis.update.end`
- `baremetal.chassis.update.error`
- `baremetal.chassis.delete.start`
- `baremetal.chassis.delete.end`
- `baremetal.chassis.delete.error`

Example of chassis CRUD notification:

```
{
 "priority": "info",
 "payload":{
     "ironic_object.namespace":"ironic",
     "ironic_object.name":"ChassisCRUDPayload",
     "ironic_object.version":"1.0",
     "ironic_object.data":{
         "created_at": "2016-04-10T10:13:03+00:00",
         "description": "bare 28",
```

(continues on next page)

[3] https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

(continued from previous page)


```
        "extra": {},
        "updated_at": "2016-04-27T21:11:03+00:00",
        "uuid": "1910f669-ce8b-43c2-b1d8-cf3d65be815e"
    }
},
"event_type":"baremetal.chassis.update.end",
"publisher_id":"ironic-api.hostname02"
}
```


List of CRUD notifications for deploy template:

- baremetal.deploy_template.create.start
- baremetal.deploy_template.create.end
- baremetal.deploy_template.create.error
- baremetal.deploy_template.update.start
- baremetal.deploy_template.update.end
- baremetal.deploy_template.update.error
- baremetal.deploy_template.delete.start
- baremetal.deploy_template.delete.end
- baremetal.deploy_template.delete.error

Example of deploy template CRUD notification:


```
{
"priority": "info",
"payload":{
    "ironic_object.namespace":"ironic",
    "ironic_object.name":"DeployTemplateCRUDPayload",
    "ironic_object.version":"1.0",
    "ironic_object.data":{
        "created_at": "2019-02-10T10:13:03+00:00",
        "extra": {},
        "name": "CUSTOM_HYPERTHREADING_ON",
        "steps": [
            {
                "interface": "bios",
                "step": "apply_configuration",
                "args": {
                    "settings": [
                        {
                            "name": "LogicalProc",
                            "value": "Enabled"
                        }
                    ]
                },
                "priority": 150
```


(continues on next page)

(continued from previous page)


```
            }
        ],
        "updated_at": "2019-02-27T21:11:03+00:00",
        "uuid": "1910f669-ce8b-43c2-b1d8-cf3d65be815e"
    }
},
"event_type":"baremetal.deploy_template.update.end",
"publisher_id":"ironic-api.hostname02"
}
```


List of CRUD notifications for node:

- baremetal.node.create.start
- baremetal.node.create.end
- baremetal.node.create.error
- baremetal.node.update.start
- baremetal.node.update.end
- baremetal.node.update.error
- baremetal.node.delete.start
- baremetal.node.delete.end
- baremetal.node.delete.error

Example of node CRUD notification:


```
{
"priority": "info",
"payload":{
    "ironic_object.namespace":"ironic",
    "ironic_object.name":"NodeCRUDPayload",
    "ironic_object.version":"1.13",
    "ironic_object.data":{
        "chassis_uuid": "db0eef9d-45b2-4dc0-94a8-fc283c01171f",
        "clean_step": None,
        "conductor_group": "",
        "console_enabled": False,
        "created_at": "2016-01-26T20:41:03+00:00",
        "deploy_step": None,
        "description": "my sample node",
        "driver": "ipmi",
        "driver_info": {
            "ipmi_address": "192.168.0.111",
            "ipmi_username": "root"},
        "extra": {},
        "inspection_finished_at": None,
        "inspection_started_at": None,
        "instance_info": {},
```


(continues on next page)

(continued from previous page)


```
        "instance_uuid": None,
        "last_error": None,
        "lessee": None,
        "maintenance": False,
        "maintenance_reason": None,
        "fault": None,
        "bios_interface": "no-bios",
        "boot_interface": "pxe",
        "console_interface": "no-console",
        "deploy_interface": "direct",
        "inspect_interface": "no-inspect",
        "management_interface": "ipmitool",
        "network_interface": "flat",
        "power_interface": "ipmitool",
        "raid_interface": "no-raid",
        "rescue_interface": "no-rescue",
        "storage_interface": "noop",
        "vendor_interface": "no-vendor",
        "name": None,
        "owner": None,
        "power_state": "power off",
        "properties": {
            "memory_mb":  4096,
            "cpu_arch":  "x86_64",
            "local_gb":  10,
            "cpus":  8},
        "protected": False,
        "protected_reason": None,
        "provision_state": "deploying",
        "provision_updated_at": "2016-01-27T20:41:03+00:00",
        "resource_class": None,
        "retired": None,
        "retired_reason": None,
        "target_power_state": None,
        "target_provision_state": "active",
        "traits": [
            "CUSTOM_TRAIT1",
            "HW_CPU_X86_VMX"],
        "updated_at": "2016-01-27T20:41:03+00:00",
        "uuid": "1be26c0b-03f2-4d2e-ae87-c02d7f33c123"
    }
},
"event_type":"baremetal.node.update.end",
"publisher_id":"ironic-api.hostname02"
}
```


List of CRUD notifications for port:

- `baremetal.port.create.start`
- `baremetal.port.create.end`

- baremetal.port.create.error
- baremetal.port.update.start
- baremetal.port.update.end
- baremetal.port.update.error
- baremetal.port.delete.start
- baremetal.port.delete.end
- baremetal.port.delete.error

Example of port CRUD notification:


```
{
 "priority": "info",
 "payload":{
     "ironic_object.namespace":"ironic",
     "ironic_object.name":"PortCRUDPayload",
     "ironic_object.version":"1.3",
     "ironic_object.data":{
         "address": "77:66:23:34:11:b7",
         "created_at": "2016-02-11T15:23:03+00:00",
         "node_uuid": "5b236cab-ad4e-4220-b57c-e827e858745a",
         "extra": {},
         "is_smartnic": True,
         "local_link_connection": {},
         "physical_network": "physnet1",
         "portgroup_uuid": "bd2f385e-c51c-4752-82d1-7a9ec2c25f24",
         "pxe_enabled": True,
         "updated_at": "2016-03-27T20:41:03+00:00",
         "uuid": "1be26c0b-03f2-4d2e-ae87-c02d7f33c123"
     }
 },
 "event_type":"baremetal.port.update.end",
 "publisher_id":"ironic-api.hostname02"
}
```


List of CRUD notifications for port group:

- baremetal.portgroup.create.start
- baremetal.portgroup.create.end
- baremetal.portgroup.create.error
- baremetal.portgroup.update.start
- baremetal.portgroup.update.end
- baremetal.portgroup.update.error
- baremetal.portgroup.delete.start
- baremetal.portgroup.delete.end
- baremetal.portgroup.delete.error

Example of portgroup CRUD notification:


```
{
 "priority": "info",
 "payload":{
     "ironic_object.namespace":"ironic",
     "ironic_object.name":"PortgroupCRUDPayload",
     "ironic_object.version":"1.0",
     "ironic_object.data":{
         "address": "11:44:32:87:61:e5",
         "created_at": "2017-01-11T11:33:03+00:00",
         "node_uuid": "5b236cab-ad4e-4220-b57c-e827e858745a",
         "extra": {},
         "mode": "7",
         "name": "portgroup-node-18",
         "properties": {},
         "standalone_ports_supported": True,
         "updated_at": "2017-01-31T11:41:07+00:00",
         "uuid": "db033a40-bfed-4c84-815a-3db26bb268bb",
     }
 },
 "event_type":"baremetal.portgroup.update.end",
 "publisher_id":"ironic-api.hostname02"
}
```


List of CRUD notifications for volume connector:

- `baremetal.volumeconnector.create.start`
- `baremetal.volumeconnector.create.end`
- `baremetal.volumeconnector.create.error`
- `baremetal.volumeconnector.update.start`
- `baremetal.volumeconnector.update.end`
- `baremetal.volumeconnector.update.error`
- `baremetal.volumeconnector.delete.start`
- `baremetal.volumeconnector.delete.end`
- `baremetal.volumeconnector.delete.error`

Example of volume connector CRUD notification:


```
{
 "priority": "info",
 "payload": {
     "ironic_object.namespace": "ironic",
     "ironic_object.name": "VolumeConnectorCRUDPayload",
     "ironic_object.version": "1.0",
     "ironic_object.data": {
        "connector_id": "iqn.2017-05.org.openstack:01:d9a51732c3f",
        "created_at": "2017-05-11T05:57:36+00:00",
```


(continues on next page)

(continued from previous page)


```
        "extra": {},
        "node_uuid": "4dbb4e69-99a8-4e13-b6e8-dd2ad4a20caf",
        "type": "iqn",
        "updated_at": "2017-05-11T08:28:58+00:00",
        "uuid": "19b9f3ab-4754-4725-a7a4-c43ea7e57360"
    }
 },
 "event_type": "baremetal.volumeconnector.update.end",
 "publisher_id":"ironic-api.hostname02"
}
```


List of CRUD notifications for volume target:

- baremetal.volumetarget.create.start
- baremetal.volumetarget.create.end
- baremetal.volumetarget.create.error
- baremetal.volumetarget.update.start
- baremetal.volumetarget.update.end
- baremetal.volumetarget.update.error
- baremetal.volumetarget.delete.start
- baremetal.volumetarget.delete.end
- baremetal.volumetarget.delete.error

Example of volume target CRUD notification:


```
{
 "priority": "info",
 "payload": {
     "ironic_object.namespace": "ironic",
     "ironic_object.version": "1.0",
     "ironic_object.name": "VolumeTargetCRUDPayload"
     "ironic_object.data": {
         "boot_index": 0,
         "created_at": "2017-05-11T09:38:59+00:00",
         "extra": {},
         "node_uuid": "4dbb4e69-99a8-4e13-b6e8-dd2ad4a20caf",
         "properties": {
             "access_mode": "rw",
             "auth_method": "CHAP"
             "auth_password": "***",
             "auth_username": "urxhQCzAKr4sjyE8DivY",
             "encrypted": false,
             "qos_specs": null,
             "target_discovered": false,
             "target_iqn": "iqn.2010-10.org.openstack:volume-f0d9b0e6-b242-
↪9105-91d4-a20331693ad8",
```


(continues on next page)

(continued from previous page)


```
            "target_lun": 1,
            "target_portal": "192.168.12.34:3260",
            "volume_id": "f0d9b0e6-b042-4105-91d4-a20331693ad8",
        },
        "updated_at": "2017-05-11T09:52:04+00:00",
        "uuid": "82a45833-9c58-4ec1-943c-2091ab10e47b",
        "volume_id": "f0d9b0e6-b242-9105-91d4-a20331693ad8",
        "volume_type": "iscsi"
    }
},
"event_type": "baremetal.volumetarget.update.end",
"publisher_id":"ironic-api.hostname02"
}
```


Node maintenance notifications

These notifications are emitted from API service when maintenance mode is changed via API service. List of maintenance notifications for a node:

- `baremetal.node.maintenance_set.start`
- `baremetal.node.maintenance_set.end`
- `baremetal.node.maintenance_set.error`

start and end notifications have INFO level, error has ERROR. Example of node maintenance notification:


```
{
"priority": "info",
"payload":{
    "ironic_object.namespace":"ironic",
    "ironic_object.name":"NodePayload",
    "ironic_object.version":"1.15",
    "ironic_object.data":{
        "clean_step": None,
        "conductor_group": "",
        "console_enabled": False,
        "created_at": "2016-01-26T20:41:03+00:00",
        "deploy_step": None,
        "description": "my sample node",
        "driver": "ipmi",
        "extra": {},
        "inspection_finished_at": None,
        "inspection_started_at": None,
        "instance_info": {},
        "instance_uuid": None,
        "last_error": None,
        "lessee": None,
        "maintenance": True,
        "maintenance_reason": "hw upgrade",
```


(continues on next page)

(continued from previous page)


```
        "fault": None,
        "bios_interface": "no-bios",
        "boot_interface": "pxe",
        "console_interface": "no-console",
        "deploy_interface": "direct",
        "inspect_interface": "no-inspect",
        "management_interface": "ipmitool",
        "network_interface": "flat",
        "power_interface": "ipmitool",
        "raid_interface": "no-raid",
        "rescue_interface": "no-rescue",
        "storage_interface": "noop",
        "vendor_interface": "no-vendor",
        "name": None,
        "owner": None,
        "power_state": "power off",
        "properties": {
            "memory_mb":  4096,
            "cpu_arch":  "x86_64",
            "local_gb":  10,
            "cpus":  8},
        "protected": False,
        "protected_reason": None,
        "provision_state": "available",
        "provision_updated_at": "2016-01-27T20:41:03+00:00",
        "resource_class": None,
        "retired": None,
        "retired_reason": None,
        "target_power_state": None,
        "target_provision_state": None,
        "traits": [
            "CUSTOM_TRAIT1",
            "HW_CPU_X86_VMX"],
        "updated_at": "2016-01-27T20:41:03+00:00",
        "uuid": "1be26c0b-03f2-4d2e-ae87-c02d7f33c123"
    }
},
"event_type":"baremetal.node.maintenance_set.start",
"publisher_id":"ironic-api.hostname02"
}
```

ironic-conductor notifications

Node console notifications

These notifications are emitted by the ironic-conductor service when conductor service starts or stops console for the node. The notification event types for a node console are:

- `baremetal.node.console_set.start`
- `baremetal.node.console_set.end`
- `baremetal.node.console_set.error`
- `baremetal.node.console_restore.start`
- `baremetal.node.console_restore.end`
- `baremetal.node.console_restore.error`

`console_set` action is used when start or stop console is initiated. The `console_restore` action is used when the console was already enabled, but a driver must restart the console because an ironic-conductor was restarted. This may also be sent when an ironic-conductor takes over a node that was being managed by another ironic-conductor. start and end notifications have INFO level, error has ERROR. Example of node console notification:


```
{
 "priority": "info",
 "payload":{
     "ironic_object.namespace":"ironic",
     "ironic_object.name":"NodePayload",
     "ironic_object.version":"1.15",
     "ironic_object.data":{
         "clean_step": None,
         "conductor_group": "",
         "console_enabled": True,
         "created_at": "2016-01-26T20:41:03+00:00",
         "deploy_step": None,
         "description": "my sample node",
         "driver": "ipmi",
         "extra": {},
         "inspection_finished_at": None,
         "inspection_started_at": None,
         "instance_info": {},
         "instance_uuid": None,
         "last_error": None,
         "lessee": None,
         "maintenance": False,
         "maintenance_reason": None,
         "fault": None,
         "bios_interface": "no-bios",
         "boot_interface": "pxe",
         "console_interface": "no-console",
         "deploy_interface": "direct",
         "inspect_interface": "no-inspect",
```


(continues on next page)

(continued from previous page)


```
        "management_interface": "ipmitool",
        "network_interface": "flat",
        "power_interface": "ipmitool",
        "raid_interface": "no-raid",
        "rescue_interface": "no-rescue",
        "storage_interface": "noop",
        "vendor_interface": "no-vendor",
        "name": None,
        "owner": None,
        "power_state": "power off",
        "properties": {
            "memory_mb":  4096,
            "cpu_arch":  "x86_64",
            "local_gb":  10,
            "cpus":  8},
        "protected": False,
        "protected_reason": None,
        "provision_state": "available",
        "provision_updated_at": "2016-01-27T20:41:03+00:00",
        "resource_class": None,
        "retired": None,
        "retired_reason": None,
        "target_power_state": None,
        "target_provision_state": None,
        "traits": [
            "CUSTOM_TRAIT1",
            "HW_CPU_X86_VMX"],
        "updated_at": "2016-01-27T20:41:03+00:00",
        "uuid": "1be26c0b-03f2-4d2e-ae87-c02d7f33c123"
    }
},
"event_type":"baremetal.node.console_set.end",
"publisher_id":"ironic-conductor.hostname01"
}
```


baremetal.node.power_set

- `baremetal.node.power_set.start` is emitted by the ironic-conductor service when it begins a power state change. It has notification level info.
- `baremetal.node.power_set.end` is emitted when ironic-conductor successfully completes a power state change task. It has notification level info.
- `baremetal.node.power_set.error` is emitted by ironic-conductor when it fails to set a nodes power state. It has notification level error. This can occur when ironic fails to retrieve the old power state prior to setting the new one on the node, or when it fails to set the power state if a change is requested.

Here is an example payload for a notification with this event type. The to_power payload field indicates the power state to which the ironic-conductor is attempting to change the node:

```
{
 "priority": "info",
 "payload":{
     "ironic_object.namespace":"ironic",
     "ironic_object.name":"NodeSetPowerStatePayload",
     "ironic_object.version":"1.15",
     "ironic_object.data":{
         "clean_step": None,
         "conductor_group": "",
         "console_enabled": False,
         "created_at": "2016-01-26T20:41:03+00:00",
         "deploy_step": None,
         "description": "my sample node",
         "driver": "ipmi",
         "extra": {},
         "inspection_finished_at": None,
         "inspection_started_at": None,
         "instance_uuid": "d6ea00c1-1f94-4e95-90b3-3462d7031678",
         "last_error": None,
         "lessee": None,
         "maintenance": False,
         "maintenance_reason": None,
         "fault": None,
         "bios_interface": "no-bios",
         "boot_interface": "pxe",
         "console_interface": "no-console",
         "deploy_interface": "direct",
         "inspect_interface": "no-inspect",
         "management_interface": "ipmitool",
         "network_interface": "flat",
         "power_interface": "ipmitool",
         "raid_interface": "no-raid",
         "rescue_interface": "no-rescue",
         "storage_interface": "noop",
         "vendor_interface": "no-vendor",
         "name": None,
         "owner": None,
         "power_state": "power off",
         "properties": {
             "memory_mb":  4096,
             "cpu_arch":  "x86_64",
             "local_gb":  10,
             "cpus":  8},
         "protected": False,
         "protected_reason": None
         "provision_state": "available",
         "provision_updated_at": "2016-01-27T20:41:03+00:00",
         "resource_class": None,
         "retired": None,
         "retired_reason": None,
```


(continues on next page)

(continued from previous page)


```
        "target_power_state": None,
        "target_provision_state": None,
        "traits": [
            "CUSTOM_TRAIT1",
            "HW_CPU_X86_VMX"],
        "updated_at": "2016-01-27T20:41:03+00:00",
        "uuid": "1be26c0b-03f2-4d2e-ae87-c02d7f33c123",
        "to_power": "power on"
    }
},
"event_type":"baremetal.node.power_set.start",
"publisher_id":"ironic-conductor.hostname01"
}
```


baremetal.node.power_state_corrected

- `baremetal.node.power_state_corrected.success` is emitted by ironic-conductor when the power state on the baremetal hardware is different from the previous known power state of the node and the database is corrected to reflect this new power state. It has notification level info.

Here is an example payload for a notification with this event_type. The from_power payload field indicates the previous power state on the node, prior to the correction:


```
{
"priority": "info",
"payload":{
    "ironic_object.namespace":"ironic",
    "ironic_object.name":"NodeCorrectedPowerStatePayload",
    "ironic_object.version":"1.15",
    "ironic_object.data":{
        "clean_step": None,
        "conductor_group": "",
        "console_enabled": False,
        "created_at": "2016-01-26T20:41:03+00:00",
        "deploy_step": None,
        "description": "my sample node",
        "driver": "ipmi",
        "extra": {},
        "inspection_finished_at": None,
        "inspection_started_at": None,
        "instance_uuid": "d6ea00c1-1f94-4e95-90b3-3462d7031678",
        "last_error": None,
        "lessee": None,
        "maintenance": False,
        "maintenance_reason": None,
        "fault": None,
        "bios_interface": "no-bios",
        "boot_interface": "pxe",
        "console_interface": "no-console",
```


(continues on next page)

(continued from previous page)


```
        "deploy_interface": "direct",
        "inspect_interface": "no-inspect",
        "management_interface": "ipmitool",
        "network_interface": "flat",
        "power_interface": "ipmitool",
        "raid_interface": "no-raid",
        "rescue_interface": "no-rescue",
        "storage_interface": "noop",
        "vendor_interface": "no-vendor",
        "name": None,
        "owner": None,
        "power_state": "power off",
        "properties": {
            "memory_mb":  4096,
            "cpu_arch":  "x86_64",
            "local_gb":  10,
            "cpus":  8},
        "protected": False,
        "protected_reason": None,
        "provision_state": "available",
        "provision_updated_at": "2016-01-27T20:41:03+00:00",
        "resource_class": None,
        "retired": None,
        "retired_reason": None,
        "target_power_state": None,
        "target_provision_state": None,
        "traits": [
            "CUSTOM_TRAIT1",
            "HW_CPU_X86_VMX"],
        "updated_at": "2016-01-27T20:41:03+00:00",
        "uuid": "1be26c0b-03f2-4d2e-ae87-c02d7f33c123",
        "from_power": "power on"
    }
 },
 "event_type":"baremetal.node.power_state_corrected.success",
 "publisher_id":"ironic-conductor.cond-hostname02"
}
```


baremetal.node.provision_set

- `baremetal.node.provision_set.start` is emitted by the ironic-conductor service when it begins a provision state transition. It has notification level INFO.
- `baremetal.node.provision_set.end` is emitted when ironic-conductor successfully completes a provision state transition. It has notification level INFO.
- `baremetal.node.provision_set.success` is emitted when ironic-conductor successfully changes provision state instantly, without any intermediate work required (example is AVAILABLE to MANAGEABLE). It has notification level INFO.

- `baremetal.node.provision_set.error` is emitted by ironic-conductor when it changes provision state as result of error event processing. It has notification level ERROR.

Here is an example payload for a notification with this event type. The previous_provision_state and previous_target_provision_state payload fields indicate a nodes provision states before state change, event is the FSM (finite state machine) event that triggered the state change:


```
{
 "priority": "info",
 "payload":{
     "ironic_object.namespace":"ironic",
     "ironic_object.name":"NodeSetProvisionStatePayload",
     "ironic_object.version":"1.16",
     "ironic_object.data":{
         "clean_step": None,
         "conductor_group": "",
         "console_enabled": False,
         "created_at": "2016-01-26T20:41:03+00:00",
         "deploy_step": None,
         "description": "my sample node",
         "driver": "ipmi",
         "driver_internal_info": {
             "is_whole_disk_image": True},
         "extra": {},
         "inspection_finished_at": None,
         "inspection_started_at": None,
         "instance_info": {},
         "instance_uuid": None,
         "last_error": None,
         "lessee": None,
         "maintenance": False,
         "maintenance_reason": None,
         "fault": None,
         "bios_interface": "no-bios",
         "boot_interface": "pxe",
         "console_interface": "no-console",
         "deploy_interface": "direct",
         "inspect_interface": "no-inspect",
         "management_interface": "ipmitool",
         "network_interface": "flat",
         "power_interface": "ipmitool",
         "raid_interface": "no-raid",
         "rescue_interface": "no-rescue",
         "storage_interface": "noop",
         "vendor_interface": "no-vendor",
         "name": None,
         "owner": None,
         "power_state": "power off",
         "properties": {
             "memory_mb":  4096,
             "cpu_arch":  "x86_64",
```


(continues on next page)

(continued from previous page)


```
            "local_gb":  10,
            "cpus":  8},
        "protected": false,
        "protected_reason": null,
        "provision_state": "deploying",
        "provision_updated_at": "2016-01-27T20:41:03+00:00",
        "resource_class": null,
        "retired": null,
        "retired_reason": null,
        "target_power_state": null,
        "target_provision_state": "active",
        "traits": [
            "CUSTOM_TRAIT1",
            "HW_CPU_X86_VMX"],
        "updated_at": "2016-01-27T20:41:03+00:00",
        "uuid": "1be26c0b-03f2-4d2e-ae87-c02d7f33c123",
        "previous_provision_state": "available",
        "previous_target_provision_state": null,
        "event": "deploy"
    }
},
"event_type":"baremetal.node.provision_set.start",
"publisher_id":"ironic-conductor.hostname01"
}
```


4.2.15 Conductor Groups

Overview

Large scale operators tend to have needs that involve creating well defined and delinated resources. In some cases, these systems may reside close by or in far away locations. Reasoning may be simple or complex, and yet is only known to the deployer and operator of the infrastructure.

A common case is the need for delineated high availability domains where it would be much more efficient to manage a datacenter in Antarctica with a conductor in Antarctica, as opposed to a conductor in New York City.

How it works

Starting in ironic 11.1, each node has a `conductor_group` field which influences how the ironic conductor calculates (and thus allocates) baremetal nodes under ironics management. This calculation is performed independently by each operating conductor and as such if a conductor has a `[conductor]conductor_group` configuration option defined in its *ironic.conf* configuration file, the conductor will then be limited to only managing nodes with a matching `conductor_group` string.

Note: Any conductor without a `[conductor]conductor_group` setting will only manage baremetal nodes without a `conductor_group` value set upon node creation. If no such conductor is present when

conductor groups are configured, node creation will fail unless a `conductor_group` is specified upon node creation.

Warning: Nodes without a `conductor_group` setting can only be managed when a conductor exists that does not have a `[conductor]conductor_group` defined. If all conductors have been migrated to use a conductor group, such nodes are effectively orphaned.

How to use

A conductor group value may be any case insensitive string up to 255 characters long which matches the `^[a-zA-Z0-9_\-\.]*$` regular expression.

1. Set the `[conductor]conductor_group` option in ironic.conf on one or more, but not all conductors:

```
[conductor]
conductor_group = OperatorDefinedString
```

2. Restart the ironic-conductor service.
3. Set the conductor group on one or more nodes:

```
baremetal node set \
    --conductor-group "OperatorDefinedString" <uuid>
```

4. As desired and as needed, remaining conductors can be updated with the first two steps. Please be mindful of the constraints covered earlier in the document related to ability to manage nodes.

4.2.16 Security Overview

While the Bare Metal service is intended to be a secure application, it is important to understand what it does and does not cover today.

Deployers must properly evaluate their use case and take the appropriate actions to secure their environment(s). This document is intended to provide an overview of what risks an operator of the Bare Metal service should be aware of. It is not intended as a How-To guide for securing a data center or an OpenStack deployment.

REST API: user roles and policy settings

Warning: This information is presently in flux as of the Wallaby release with the implementation of `Secure RBAC` where `system` and `project` scoped requests are able to be parsed and default access controls support a delineation of roles and responsibilities through the roles. Please see *Secure RBAC*.

Beginning with the Newton (6.1.0) release, the Bare Metal service allows operators significant control over API access:

- Access may be restricted to each method (GET, PUT, etc) for each REST resource. Defaults are provided with the release and defined in code.
- Access may be divided between an administrative role with full access and observer role with read-only access. By default, these roles are assigned the names `baremetal_admin` and `baremetal_observer`, respectively.
- By default, passwords and instance secrets are hidden in `driver_info` and `instance_info`, respectively. In case of debugging or diagnosing, the behavior can be overridden by changing the policy file. To allow password in `driver_info` unmasked for users with administrative privileges, apply following changes to policy configuration file:

```
"show_password": "rule:is_admin"
```

  And restart the Bare Metal API service to take effect. Please check *Policies* for more details.

Prior to the Newton (6.1.0) release, the Bare Metal service only supported two policy options:

- API access may be secured by a simple policy rule: users with administrative privileges may access all API resources, whereas users without administrative privileges may only access public API resources.
- Passwords contained in the `driver_info` field may be hidden from all API responses with the `show_password` policy setting. This defaults to always hide passwords, regardless of the users role. You can override it with policy configuration as described above.

Multi-tenancy

There are two aspects of multitenancy to consider when evaluating a deployment of the Bare Metal Service: interactions between tenants on the network, and actions one tenant can take on a machine that will affect the next tenant.

Network Interactions

Interactions between tenants workloads running simultaneously on separate servers include, but are not limited to: IP spoofing, packet sniffing, and network man-in-the-middle attacks.

By default, the Bare Metal service provisions all nodes on a flat network, and does not take any precautions to avoid or prevent interaction between tenants. This can be addressed by integration with the OpenStack Identity, Compute, and Networking services, so as to provide tenant-network isolation. Additional documentation on network multi-tenancy is available.

Lingering Effects

Interactions between tenants placed sequentially on the same server include, but are not limited to: changes in BIOS settings, modifications to firmware, or files left on disk or peripheral storage devices (if these devices are not erased between uses).

By default, the Bare Metal service will erase (clean) the local disk drives during the cleaning phase, after deleting an instance. It *does not* reset BIOS or reflash firmware or peripheral devices. This can be addressed through customizing the utility ramdisk used during the cleaning phase. See details in the *Firmware security* section.

Firmware security

When the Bare Metal service deploys an operating system image to a server, that image is run natively on the server without virtualization. Any user with administrative access to the deployed instance has administrative access to the underlying hardware.

Most servers default settings do not prevent a privileged local user from gaining direct access to hardware devices. Such a user could modify device or firmware settings, and potentially flash new firmware to the device, before deleting their instance and allowing the server to be allocated to another user.

If the `[conductor]/automated_clean` configuration option is enabled (and the `[deploy]/erase_devices_priority` configuration option is not zero), the Bare Metal service will securely erase all local disk devices within a machine during instance deletion. However, the service does not ship with any code that will validate the integrity of, or make any modifications to, system or device firmware or firmware settings.

Operators are encouraged to write their own hardware manager plugins for the `ironic-python-agent` ramdisk. This should include custom `clean steps` that would be run during the *Node cleaning* process, as part of Node de-provisioning. The `clean steps` would perform the specific actions necessary within that environment to ensure the integrity of each servers firmware.

Ideally, an operator would work with their hardware vendor to ensure that proper firmware security measures are put in place ahead of time. This could include:

- installing signed firmware for BIOS and peripheral devices
- using a TPM (Trusted Platform Module) to validate signatures at boot time
- booting machines in *UEFI secure boot mode*, rather than BIOS mode, to validate kernel signatures
- disabling local (in-band) access from the host OS to the management controller (BMC)
- disabling modifications to boot settings from the host OS

Additional references:

- *Node cleaning*

UEFI secure boot mode

Some hardware types support turning UEFI secure boot dynamically when deploying an instance. Currently these are *iLO driver*, *iRMC driver* and *Redfish driver*.

Support for the UEFI secure boot is declared by adding the `secure_boot` capability in the `capabilities` parameter in the `properties` field of a node. `secure_boot` is a boolean parameter and takes value as `true` or `false`.

To enable `secure_boot` on a node add it to `capabilities`:

```
baremetal node set <node> --property capabilities='secure_boot:true'
```

Alternatively use *Hardware Inspection* to automatically populate the secure boot capability.

Warning: UEFI secure boot only works in UEFI boot mode, see *Boot mode support* for how to turn it on and off.

Compatible images

Use element `ubuntu-signed` or `fedora` to build signed deploy ISO and user images with diskimage-builder.

The below command creates files named cloud-image-boot.iso, cloud-image.initrd, cloud-image.vmlinuz and cloud-image.qcow2 in the current working directory:

```
cd <path-to-diskimage-builder>
./bin/disk-image-create -o cloud-image ubuntu-signed baremetal iso
```

Ensure the public key of the signed image is loaded into bare metal to deploy signed images.

Enabling with OpenStack Compute

Nodes having `secure_boot` set to `true` may be requested by adding an `extra_spec` to the nova flavor:

```
openstack flavor set <flavor> --property capabilities:secure_boot="true"
openstack server create --flavor <flavor> --image <image> instance-1
```

If `capabilities` is used in `extra_spec` as above, nova scheduler (`ComputeCapabilitiesFilter`) will match only ironic nodes which have the `secure_boot` set appropriately in `properties/capabilities`. It will filter out rest of the nodes.

The above facility for matching in nova can be used in heterogeneous environments where there is a mix of machines supporting and not supporting UEFI secure boot, and operator wants to provide a choice to the user regarding secure boot. If the flavor doesnt contain `secure_boot` then nova scheduler will not consider secure boot mode as a placement criteria, hence user may get a secure boot capable machine that matches with user specified flavors but deployment would not use its secure boot capability. Secure boot deploy would happen only when it is explicitly specified through flavor.

Enabling standalone

To request secure boot for an instance in standalone mode (without OpenStack Compute), you need to add the capability directly to the nodes `instance_info`:

```
baremetal node set <node> --instance-info capabilities='{"secure_boot": "true
↪"}'
```

Other considerations

Internal networks

Access to networks which the Bare Metal service uses internally should be prohibited from outside. These networks are the ones used for management (with the nodes BMC controllers), provisioning, cleaning (if used) and rescuing (if used).

This can be done with physical or logical network isolation, traffic filtering, etc.

Management interface technologies

Some nodes support more than one management interface technology (vendor and IPMI for example). If you use only one modern technology for out-of-band node access, it is recommended that you disable IPMI since the IPMI protocol is not secure. If IPMI is enabled, in most cases a local OS administrator is able to work in-band with IPMI settings without specifying any credentials, as this is a DCMI specification requirement.

Tenant network isolation

If you use tenant network isolation, services (TFTP or HTTP) that handle the nodes boot files should serve requests only from the internal networks that are used for the nodes being deployed and cleaned.

TFTP protocol does not support per-user access control at all.

For HTTP, there is no generic and safe way to transfer credentials to the node.

Also, tenant network isolation is not intended to work with network-booting a node by default, once the node has been provisioned.

API endpoints for RAM disk use

There are two (unauthorized) endpoints in the Bare Metal API that are intended for use by the ironic-python-agent RAM disk. They are not intended for public use.

These endpoints can potentially cause security issues. Access to these endpoints from external or untrusted networks should be prohibited. An easy way to do this is to:

- set up two groups of API services: one for external requests, the second for deploy RAM disks requests.
- to disable unauthorized access to these endpoints in the (first) API services group that serves external requests, the following lines should be added to the policy.yaml file:

```
# Send heartbeats from IPA ramdisk
"baremetal:node:ipa_heartbeat": "rule:is_admin"

# Access IPA ramdisk functions
"baremetal:driver:ipa_lookup": "rule:is_admin"
```

4.2.17 Troubleshooting Ironic

Nova returns No valid host was found Error

Sometimes Nova Conductor log file nova-conductor.log or a message returned from Nova API contains the following error:

```
NoValidHost: No valid host was found. There are not enough hosts available.
```

No valid host was found means that the Nova Scheduler could not find a bare metal node suitable for booting the new instance.

This in turn usually means some mismatch between resources that Nova expects to find and resources that Ironic advertised to Nova.

A few things should be checked in this case:

1. Make sure that enough nodes are in `available` state, not in maintenance mode and not already used by an existing instance. Check with the following command:

   ```
   baremetal node list --provision-state available --no-maintenance --
   ↪unassociated
   ```

   If this command does not show enough nodes, use generic `baremetal node list` to check other nodes. For example, nodes in `manageable` state should be made available:

   ```
   baremetal node provide <IRONIC NODE>
   ```

   The Bare metal service automatically puts a node in maintenance mode if there are issues with accessing its management interface. See *Power fault and recovery* for details.

   The `node validate` command can be used to verify that all required fields are present. The following command should not return anything:

   ```
   baremetal node validate <IRONIC NODE> | grep -E '(power|management)\
   ↪W*False'
   ```

   Maintenance mode will be also set on a node if automated cleaning has failed for it previously.

2. Make sure that you have Compute services running and enabled:

   ```
   $ openstack compute service list --service nova-compute
   +----+--------------+-------------+------+---------+-------+--------------
   ↪--------------+
   | ID | Binary       | Host        | Zone | Status  | State | Updated At  ␣
   ↪              |
   +----+--------------+-------------+------+---------+-------+--------------
   ↪--------------+
   |  7 | nova-compute | example.com | nova | enabled | up    | 2017-09-
   ↪04T13:14:03.000000 |
   +----+--------------+-------------+------+---------+-------+--------------
   ↪--------------+
   ```

   By default, a Compute service is disabled after 10 consecutive build failures on it. This is to ensure that new build requests are not routed to a broken Compute service. If it is the case, make sure to fix the source of the failures, then re-enable it:

   ```
   openstack compute service set --enable <COMPUTE HOST> nova-compute
   ```

3. Starting with the Pike release, check that all your nodes have the `resource_class` field set using the following command:

   ```
   baremetal node list --fields uuid name resource_class
   ```

   Then check that the flavor(s) are configured to request these resource classes via their properties:

```
openstack flavor show <FLAVOR NAME> -f value -c properties
```

For example, if your node has resource class `baremetal-large`, it will be matched by a flavor with property `resources:CUSTOM_BAREMETAL_LARGE` set to 1. See *Create flavors for use with the Bare Metal service* for more details on the correct configuration.

4. Upon scheduling, Nova will query the Placement API service for the available resource providers (in the case of Ironic: nodes with a given resource class). If placement does not have any allocation candidates for the requested resource class, the request will result in a No valid host was found error. It is hence sensible to check if Placement is aware of resource providers (nodes) for the requested resource class with:

```
$ openstack allocation candidate list --resource CUSTOM_BAREMETAL_LARGE='1
↪'
+---+------------------------------+--------------------------------------
↪+------------------------------+
| # | allocation                   | resource provider                    ␣
↪| inventory used/capacity      |
+---+------------------------------+--------------------------------------
↪+------------------------------+
| 1 | CUSTOM_BAREMETAL_LARGE=1     | 2f7b9c69-c1df-4e40-b94e-5821a4ea0453␣
↪| CUSTOM_BAREMETAL_LARGE=0/1    |
+---+------------------------------+--------------------------------------
↪+------------------------------+
```

For Ironic, the resource provider is the UUID of the available Ironic node. If this command returns an empty list (or does not contain the targeted resource provider), the operator needs to understand first, why the resource tracker has not reported this provider to placement. Potential explanations include:

- the resource tracker cycle has not finished yet and the resource provider will appear once it has (the time to finish the cycle scales linearly with the number of nodes the corresponding `nova-compute` service manages);
- the node is in a state where the resource tracker does not consider it to be eligible for scheduling, e.g. when the node has `maintenance` set to `True`; make sure the target nodes are in `available` and `maintenance` is `False`;

5. If you do not use scheduling based on resource classes, then the nodes properties must have been set either manually or via inspection. For each node with `available` state check that the `properties` JSON field has valid values for the keys `cpus`, `cpu_arch`, `memory_mb` and `local_gb`. Example of valid properties:

```
$ baremetal node show <IRONIC NODE> --fields properties
+------------+------------------------------------------------------------
↪------------------------+
| Property   | Value                                                      ␣
↪                        |
+------------+------------------------------------------------------------
↪------------------------+
| properties | {u'memory_mb': u'8192', u'cpu_arch': u'x86_64', u'local_gb
↪': u'41', u'cpus': u'4'} |
```

(continues on next page)

(continued from previous page)

```
+------------+-------------------------------------------------------------
↪-----------------------+
```

Warning: If youre using exact match filters in the Nova Scheduler, make sure the flavor and the node properties match exactly.

6. The Nova flavor that you are using does not match any properties of the available Ironic nodes. Use

```
openstack flavor show <FLAVOR NAME>
```

to compare. The extra specs in your flavor starting with `capability:` should match ones in `node.properties['capabilities']`.

Note: The format of capabilities is different in Nova and Ironic. E.g. in Nova flavor:

```
$ openstack flavor show <FLAVOR NAME> -c properties
+------------+----------------------------------+
| Field      | Value                            |
+------------+----------------------------------+
| properties | capabilities:boot_option='local' |
+------------+----------------------------------+
```

But in Ironic node:

```
$ baremetal node show <IRONIC NODE> --fields properties
+------------+--------------------------------------+
| Property   | Value                                |
+------------+--------------------------------------+
| properties | {u'capabilities': u'boot_option:local'} |
+------------+--------------------------------------+
```

---

7. After making changes to nodes in Ironic, it takes time for those changes to propagate from Ironic to Nova. Check that

```
openstack hypervisor stats show
```

correctly shows total amount of resources in your system. You can also check `openstack hypervisor show <IRONIC NODE>` to see the status of individual Ironic nodes as reported to Nova.

8. Figure out which Nova Scheduler filter ruled out your nodes. Check the `nova-scheduler` logs for lines containing something like:

```
Filter ComputeCapabilitiesFilter returned 0 hosts
```

The name of the filter that removed the last hosts may give some hints on what exactly was not matched. See Nova filters documentation for more details.

9. If none of the above helped, check Ironic conductor log carefully to see if there are any conductor-related errors which are the root cause for No valid host was found. If there are any Error in deploy of node <IRONIC-NODE-UUID>: [Errno 28] error messages in Ironic conductor log, it means the conductor run into a special error during deployment. So you can check the log carefully to fix or work around and then try again.

Patching the Deploy Ramdisk

When debugging a problem with deployment and/or inspection you may want to quickly apply a change to the ramdisk to see if it helps. Of course you can inject your code and/or SSH keys during the ramdisk build (depends on how exactly youve built your ramdisk). But its also possible to quickly modify an already built ramdisk.

Create an empty directory and unpack the ramdisk content there:

```
$ mkdir unpack
$ cd unpack
$ gzip -dc /path/to/the/ramdisk | cpio -id
```

The last command will result in the whole Linux file system tree unpacked in the current directory. Now you can modify any files you want. The actual location of the files will depend on the way youve built the ramdisk.

Note: On a systemd-based system you can use the `systemd-nspawn` tool (from the `systemd-container` package) to create a lightweight container from the unpacked filesystem tree:

```
$ sudo systemd-nspawn --directory /path/to/unpacked/ramdisk/ /bin/bash
```

This will allow you to run commands within the filesystem, e.g. use package manager. If the ramdisk is also systemd-based, and you have login credentials set up, you can even boot a real ramdisk enviroment with

```
$ sudo systemd-nspawn --directory /path/to/unpacked/ramdisk/ --boot
```

After youve done the modifications, pack the whole content of the current directory back:

```
$ find . | cpio -H newc -o | gzip -c > /path/to/the/new/ramdisk
```

Note: You dont need to modify the kernel (e.g. `tinyipa-master.vmlinuz`), only the ramdisk part.

API Errors

The *debug_tracebacks_in_api* config option may be set to return tracebacks in the API response for all 4xx and 5xx errors.

Retrieving logs from the deploy ramdisk

When troubleshooting deployments (specially in case of a deploy failure) its important to have access to the deploy ramdisk logs to be able to identify the source of the problem. By default, Ironic will retrieve the logs from the deploy ramdisk when the deployment fails and save it on the local filesystem at `/var/log/ironic/deploy`.

To change this behavior, operators can make the following changes to `/etc/ironic/ironic.conf` under the `[agent]` group:

- `deploy_logs_collect`: Whether Ironic should collect the deployment logs on deployment. Valid values for this option are:
  - `on_failure` (**default**): Retrieve the deployment logs upon a deployment failure.
  - `always`: Always retrieve the deployment logs, even if the deployment succeed.
  - `never`: Disable retrieving the deployment logs.
- `deploy_logs_storage_backend`: The name of the storage backend where the logs will be stored. Valid values for this option are:
  - `local` (**default**): Store the logs in the local filesystem.
  - `swift`: Store the logs in Swift.
- `deploy_logs_local_path`: The path to the directory where the logs should be stored, used when the `deploy_logs_storage_backend` is configured to `local`. By default logs will be stored at **/var/log/ironic/deploy**.
- `deploy_logs_swift_container`: The name of the Swift container to store the logs, used when the deploy_logs_storage_backend is configured to swift. By default **ironic_deploy_logs_container**.
- `deploy_logs_swift_days_to_expire`: Number of days before a log object is marked as expired in Swift. If None, the logs will be kept forever or until manually deleted. Used when the deploy_logs_storage_backend is configured to swift. By default **30** days.

When the logs are collected, Ironic will store a *tar.gz* file containing all the logs according to the `deploy_logs_storage_backend` configuration option. All log objects will be named with the following pattern:

```
<node>[_<instance-uuid>]_<timestamp yyyy-mm-dd-hh:mm:ss>.tar.gz
```

Note: The *instance_uuid* field is not required for deploying a node when Ironic is configured to be used in standalone mode. If present it will be appended to the name.

Accessing the log data

When storing in the local filesystem

When storing the logs in the local filesystem, the log files can be found at the path configured in the `deploy_logs_local_path` configuration option. For example, to find the logs from the node `5e9258c4-cfda-40b6-86e2-e192f523d668`:

```
$ ls /var/log/ironic/deploy | grep 5e9258c4-cfda-40b6-86e2-e192f523d668
5e9258c4-cfda-40b6-86e2-e192f523d668_88595d8a-6725-4471-8cd5-c0f3106b6898_
↪2016-08-08-13:52:12.tar.gz
5e9258c4-cfda-40b6-86e2-e192f523d668_db87f2c5-7a9a-48c2-9a76-604287257c1b_
↪2016-08-08-14:07:25.tar.gz
```

Note: When saving the logs to the filesystem, operators may want to enable some form of rotation for the logs to avoid disk space problems.

When storing in Swift

When using Swift, operators can associate the objects in the container with the nodes in Ironic and search for the logs for the node `5e9258c4-cfda-40b6-86e2-e192f523d668` using the **prefix** parameter. For example:

```
$ swift list ironic_deploy_logs_container -p 5e9258c4-cfda-40b6-86e2-
↪e192f523d668
5e9258c4-cfda-40b6-86e2-e192f523d668_88595d8a-6725-4471-8cd5-c0f3106b6898_
↪2016-08-08-13:52:12.tar.gz
5e9258c4-cfda-40b6-86e2-e192f523d668_db87f2c5-7a9a-48c2-9a76-604287257c1b_
↪2016-08-08-14:07:25.tar.gz
```

To download a specific log from Swift, do:

```
$ swift download ironic_deploy_logs_container "5e9258c4-cfda-40b6-86e2-
↪e192f523d668_db87f2c5-7a9a-48c2-9a76-604287257c1b_2016-08-08-14:07:25.tar.gz
↪"
5e9258c4-cfda-40b6-86e2-e192f523d668_db87f2c5-7a9a-48c2-9a76-604287257c1b_
↪2016-08-08-14:07:25.tar.gz [auth 0.341s, headers 0.391s, total 0.391s, 0.
↪531 MB/s]
```

The contents of the log file

The log is just a `.tar.gz` file that can be extracted as:

```
$ tar xvf <file path>
```

The contents of the file may differ slightly depending on the distribution that the deploy ramdisk is using:

- For distributions using `systemd` there will be a file called **journal** which contains all the system logs collected via the `journalctl` command.
- For other distributions, the ramdisk will collect all the contents of the `/var/log` directory.

For all distributions, the log file will also contain the output of the following commands (if present): `ps`, `df`, `ip addr` and `iptables`.

Heres one example when extracting the content of a log file for a distribution that uses `systemd`:

```
$ tar xvf 5e9258c4-cfda-40b6-86e2-e192f523d668_88595d8a-6725-4471-8cd5-
↪c0f3106b6898_2016-08-08-13:52:12.tar.gz
df
ps
journal
ip_addr
iptables
```

DHCP during PXE or iPXE is inconsistent or unreliable

This can be caused by the spanning tree protocol delay on some switches. The delay prevents the switch port moving to forwarding mode during the nodes attempts to PXE, so the packets never make it to the DHCP server. To resolve this issue you should set the switch port that connects to your baremetal nodes as an edge or PortFast type port. Configured in this way the switch port will move to forwarding mode as soon as the link is established. An example on how to do that for a Cisco Nexus switch is:

```
$ config terminal
$ (config) interface eth1/11
$ (config-if) spanning-tree port type edge
```

Why does X issue occur when I am using LACP bonding with iPXE?

If you are using iPXE, an unfortunate aspect of its design and interaction with networking is an automatic response as a Link Aggregation Control Protocol (or LACP) peer to remote switches. iPXE does this for only the single port which is used for network booting.

In theory, this may help establish the port link-state faster with some switch vendors, but the official reasoning as far as the Ironic Developers are aware is not documented for iPXE. The end result of this is that once iPXE has stopped responding to LACP messages from the peer port, which occurs as part of the process of booting a ramdisk and iPXE handing over control to a full operating-system, switches typically begin a timer to determine how to handle the failure. This is because, depending on the mode of LACP, this can be interpreted as a switch or network fabric failure.

This may demonstrate as any number of behaviors or issues from ramdisks finding they are unable to acquire DHCP addresses over the network interface to downloads abruptly stalling, to even minor issues such as LLDP port data being unavailable in introspection.

Overall:

- Ironics agent doesnt officially support LACP and the Ironic community generally believes this may cause more problems than it would solve. During the Victoria development cycle, we added retry logic for most actions in an attempt to navigate the worst-known default hold-down timers to help ensure a deployment does not fail due to a short-lived transitory network connectivity failure in the form of a switch port having moved to a temporary blocking state. Where applicable and possible, many of these patches have been backported to supported releases. These patches also require that the switchport has an eventual fallback to a non-bonded mode. If the port remains in a blocking state, then traffic will be unable to flow and the deployment is likely to time out.
- If you must use LACP, consider `passive` LACP negotiation settings in the network switch as opposed to `active`. The difference being with passive the connected workload is likely a server where it should likely request the switch to establish the Link Aggregate. This is instead of being treated as if its possibly another switch.
- Consult your switch vendors support forums. Some vendors have recommended port settings for booting machines using iPXE with their switches.

IPMI errors

When working with IPMI, several settings need to be enabled depending on vendors.

Enable IPMI over LAN

Machines may not have IPMI access over LAN enabled by default. This could cause the IPMI port to be unreachable through ipmitool, as shown:

```
$ ipmitool -I lan -H ipmi_host -U ipmi_user -P ipmi_pass chassis power status
Error: Unable to establish LAN session
```

To fix this, enable *IPMI over lan* setting using your BMC tool or web app.

Troubleshooting lanplus interface

When working with lanplus interfaces, you may encounter the following error:

```
$ ipmitool -I lanplus -H ipmi_host -U ipmi_user -P ipmi_pass power status
Error in open session response message : insufficient resources for session
Error: Unable to establish IPMI v2 / RMCP+ session
```

To fix that issue, please enable *RMCP+ Cipher Suite3 Configuration* setting using your BMC tool or web app.

Why are my nodes stuck in a -ing state?

The Ironic conductor uses states ending with `ing` as a signifier that the conductor is actively working on something related to the node.

Often, this means there is an internal lock or `reservation` set on the node and the conductor is downloading, uploading, or attempting to perform some sort of Input/Output operation - see *Why does API return Node is locked by host?* for details.

In the case the conductor gets stuck, these operations should timeout, but there are cases in operating systems where operations are blocked until completion. These sorts of operations can vary based on the specific environment and operating configuration.

What can cause these sorts of failures?

Typical causes of such failures are going to be largely rooted in the concept of `iowait`, either in the form of downloading from a remote host or reading or writing to the disk of the conductor. An operator can use the iostat tool to identify the percentage of CPU time spent waiting on storage devices.

The fields that will be particularly important are the `iowait`, `await`, and `tps` ones, which can be read about in the `iostat` manual page.

In the case of network file systems, for backing components such as image caches or distributed `tftpboot` or `httpboot` folders, IO operations failing on these can, depending on operating system and underlying client settings, cause threads to be stuck in a blocking wait state, which is realistically undetectable short the operating system logging connectivity errors or even lock manager access errors.

For example with nfs, the underlying client recovery behavior, in terms of `soft`, `hard`, `softreval`, `nosoftreval`, will largely impact this behavior, but also NFS server settings can impact this behavior. A solid sign that this is a failure, is when an `ls /path/to/nfs` command hangs for a period of time. In such cases, the Storage Administrator should be consulted and network connectivity investigated for errors before trying to recover to proceed.

The bad news for IO related failures

If the node has a populated `reservation` field, and has not timed out or proceeded to a `fail` state, then the conductor process will likely need to be restarted. This is because the worker thread is hung with-in the conductor.

Manual intervention with-in Ironics database is *not* advised to try and un-wedge the machine in this state, and restarting the conductor is encouraged.

Note: Ironics conductor, upon restart, clears reservations for nodes which were previously managed by the conductor before restart.

If a distributed or network file system is in use, it is highly recommended that the operating system of the node running the conductor be rebooted as the running conductor may not even be able to exit in the state of an IO failure, again dependent upon site and server configuration.

File Size != Disk Size

An easy to make misconception is that a 2.4 GB file means that only 2.4 GB is written to disk. But if that files virtual size is 20 GB, or 100 GB things can become very problematic and extend the amount of time the node spends in `deploying` and `deploy wait` states.

Again, these sorts of cases will depend upon the exact configuration of the deployment, but hopefully these are areas where these actions can occur.

- Conversion to raw image files upon download to the conductor, from the `[DEFAULT]force_raw_images` option. Users using Glance may also experience issues here as the conductor will cache the image to be written which takes place when the `[agent]image_download_source` is set to `http` instead of `swift`.

Note: The QCOW2 image conversion utility does consume quite a bit of memory when converting images or writing them to the end storage device. This is because the files are not sequential in nature, and must be re-assembled from an internal block mapping. Internally Ironic limits this to 1GB of RAM. Operators performing large numbers of deployments may wish to disable raw images in these sorts of cases in order to minimize the conductor becoming a limiting factor due to memory and network IO.

Why are my nodes stuck in a wait state?

The Ironic conductor uses states containing `wait` as a signifier that the conductor is waiting for a callback from another component, such as the Ironic Python Agent or the Inspector. If this feedback does not arrive, the conductor will time out and the node will eventually move to a `failed` state. Depending on the configuration and the circumstances, however, a node can stay in a `wait` state for a long time or even never time out. The list of such wait states includes:

- `clean wait` for cleaning,
- `inspect wait` for introspection,
- `rescue wait` for rescueing, and
- `wait call-back` for deploying.

Communication issues between the conductor and the node

One of the most common issues when nodes seem to be stuck in a wait state occur when the node never received any instructions or does not react as expected: the conductor moved the node to a wait state but the node will never call back. Examples include wrong ciphers which will make ipmitool get stuck or BMCs in a state where they accept commands, but dont do the requested task (or only a part of it, like shutting off, but not starting). It is useful in these cases to see via a ping or the console if and which action the node is performing. If the node does not seem to react to the requests sent be the conductor, it may be worthwhile to try the corresponding action out-of-band, e.g. confirm that power on/off commands work when directly sent to the BMC. The section on *IPMI errors*. above gives some additional points to check. In some situations, a BMC reset may be necessary.

Ironic Python Agent stuck

Nodes can also get remain in a wait state when the component the conductor is waiting for gets stuck, e.g. when a hardware manager enters a loop or is waiting for an event that is never happening. In these cases, it might be helpful to connect to the IPA and inspect its logs, see the trouble shooting guide of the ironic-python-agent (IPA) on how to do this.

Stopping the operation

Cleaning, inspection and rescuing can be stopped while in `clean wait`, `inspect wait` and `rescue wait` states using the `abort` command. It will move the node to the corresponding failure state (`clean failed`, `inspect failed` or `rescue failed`):

```
baremetal node abort <node>
```

Deploying can be aborted while in the `wait call-back` state by starting an undeploy (normally resulting in cleaning):

```
baremetal node undeploy <node>
```

See *Bare Metal State Machine* for more details.

Note: Since the Bare Metal service is not doing anything actively in waiting states, the nodes are not moved to failed states on conductor restart.

Deployments fail with failed to update MAC address

The design of the integration with the Networking service (neutron) is such that once virtual ports have been created in the API, their MAC address must be updated in order for the DHCP server to be able to appropriately reply.

This can sometimes result in errors being raised indicating that the MAC address is already in use. This is because at some point in the past, a virtual interface was orphaned either by accident or by some unexpected glitch, and a previous entry is still present in Neutron.

This error looks something like this when reported in the ironic-conductor log output.:

> Failed to update MAC address on Neutron port 305beda7-0dd0-4fec-b4d2-78b7aa4e8e6a.: MacAddressInUseClient: Unable to complete operation for network 1e252627-6223-4076-a2b9-6f56493c9bac. The mac address 52:54:00:7c:c4:56 is in use.

Because we have no idea about this entry, we fail the deployment process as we cant make a number of assumptions in order to attempt to automatically resolve the conflict.

How did I get here?

Originally this was a fairly easy issue to encounter. The retry logic path which resulted between the Orchestration (heat) and Compute (nova) services, could sometimes result in additional un-necessary ports being created.

Bugs of this class have been largely resolved since the Rocky development cycle. Since then, the way this can become encountered is due to Networking (neutron) VIF attachments not being removed or deleted prior to deleting a port in the Bare Metal service.

Ultimately, the key of this is that the port is being deleted. Under most operating circumstances, there really is no need to delete the port, and VIF attachments are stored on the port object, so deleting the port *CAN* result in the VIF not being cleaned up from Neutron.

Under normal circumstances, when deleting ports, a node should be in a stable state, and the node should not be provisioned. If the `baremetal port delete` command fails, this may indicate that a known VIF is still attached. Generally if they are transitory from cleaning, provisioning, rescuing, or even inspection, getting the node to the `available` state wil unblock your delete operation, that is unless there is a tenant VIF attahment. In that case, the vif will need to be removed from with-in the Bare Metal service using the `baremetal node vif detach` command.

A port can also be checked to see if there is a VIF attachment by consulting the ports `internal_info` field.

Warning: The `maintenance` flag can be used to force the nodes port to be deleted, however this will disable any check that would normally block the user from issuing a delete and accidently orphaning the VIF attachment record.

How do I resolve this?

Generally, you need to identify the port with the offending MAC address. Example:

```
$ openstack port list --mac-address 52:54:00:7c:c4:56
```

From the commands output, you should be able to identify the `id` field. Using that, you can delete the port. Example:

```
$ openstack port delete <id>
```

Warning: Before deleting a port, you should always verify that it is no longer in use or no longer seems applicable/operable. If multiple deployments of the Bare Metal service with a single Neutron, the possibility that a inventory typo, or possibly even a duplicate MAC address exists, which could also produce the same basic error message.

My test VM image does not deploy mount point does not exist

What is likely occuring

The image attempting to be deployed likely is a partition image where the file system that the user wishes to boot from lacks the required folders, such as `/dev` and `/proc`, which are required to install a bootloader for a Linux OS image

It should be noted that similar errors can also occur with whole disk images where we are attempting to setup the UEFI bootloader configuration. That being said, in this case, the image is likely invalid or contains an unexpected internal structure.

Users performing testing may choose something that they believe will work based on it working for virtual machines. These images are often attractive for testing as they are generic and include basic support for establishing networking and possibly installing user keys. Unfortunately, these images often lack drivers and firmware required for many different types of physical hardware which makes using them very problematic. Additionally, images such as Cirros do not have any contents in the root filesystem (i.e. an empty filesystem), as they are designed for the `ramdisk` to write the contents to disk upon boot.

How do I not encounter this issue?

We generally recommend using diskimage-builder or vendor supplied images. Centos, Ubuntu, Fedora, and Debian all publish operating system images which do generally include drivers and firmware for physical hardware. Many of these published cloud images, also support auto-configuration of networking AND population of user keys.

Issues with autoconfigured TLS

These issues will manifest as an error in `ironic-conductor` logs looking similar to (lines are wrapped for readability):

```
ERROR ironic.drivers.modules.agent_client [-]
Failed to connect to the agent running on node d7c322f0-0354-4008-92b4-
↪f49fb2201001
for invoking command clean.get_clean_steps. Error:
HTTPSConnectionPool(host='192.168.123.126', port=9999): Max retries exceeded␣
↪with url:
/v1/commands/?wait=true&agent_token=<token> (Caused by
SSLError(SSLError(1, '[SSL: CERTIFICATE_VERIFY_FAILED] certificate verify␣
↪failed (_ssl.c:897)'),)):
requests.exceptions.SSLError: HTTPSConnectionPool(host='192.168.123.126',␣
↪port=9999):
Max retries exceeded with url: /v1/commands/?wait=true&agent_token=<token>
(Caused by SSLError(SSLError(1, '[SSL: CERTIFICATE_VERIFY_FAILED] certificate␣
↪verify failed (_ssl.c:897)'),))
```

The cause of the issue is that the Bare Metal service cannot access the ramdisk with the TLS certificate provided by the ramdisk on first heartbeat. You can inspect the stored certificate in `/var/lib/ironic/certificates/<node>.crt`.

You can try connecting to the ramdisk using the IP address in the log message:

```
curl -vL https://<IP address>:9999/v1/commands \
    --cacert /var/lib/ironic/certificates/<node UUID>.crt
```

You can get the detailed information about the certificate using openSSL:

```
openssl x509 -text -noout -in /var/lib/ironic/certificates/<node UUID>.crt
```

Clock skew

One possible source of the problem is a discrepancy between the hardware clock on the node and the time on the machine with the Bare Metal service. It can be detected by comparing the `Not Before` field in the `openssl` output with the timestamp of a log message.

The recommended solution is to enable the NTP support in ironic-python-agent by passing the `ipa-ntp-server` argument with an address of an NTP server reachable by the node.

If it is not possible, you need to ensure the correct hardware time on the machine. Keep in mind a potential issue with timezones: an ability to store timezone in hardware is pretty recent and may not be available. Since ironic-python-agent is likely operating in UTC, the hardware clock should also be set in UTC.

Note: Microsoft Windows uses local time by default, so a machine that has previously run Windows will likely have wrong time.

I changed ironic.conf, and now I cant edit my nodes.

Whenever a node is created in ironic, default interfaces are identified as part of driver composition. This maybe sourced from explicit default values which have been set in `ironic.conf` or by the interface order for the enabled interfaces list. The result of this is that the `ironic-conductor` cannot spawn a `task` using the composed driver, as a portion of the driver is no longer enabled. This makes it difficult to edit or update the node if the settings have been changed.

For example, with networking interfaces, if you have `default_network_interface=neutron` or `enabled_network_interfaces=neutron,flat` in your `ironic.conf`, nodes would have been created with the `neutron` network interface.

This is because `default_network_interface` overrides the setting for new nodes, and that setting is **saved** to the database nodes table.

Similarly, the order of `enabled_network_interfaces` takes priority, and the first entry in the list is generally set to the default for the node upon creation, and that record is **saved** to the database nodes table.

The only case where driver composition does *not* calculate a default is if an explicit value is provided upon the creation of the node.

Example failure

A node in this state, when the `network_interface` was saved as `neutron`, yet the `neutron` interface is no longer enabled will fail basic state transition requests:

```
$ baremetal node manage 7164efca-37ab-1213-1112-b731cf795a5a
Could not find the following interface in the 'ironic.hardware.interfaces.
↪network' entrypoint: neutron. Valid interfaces are ['flat']. (HTTP 400)
```

How to fix this?

Revert the changes you made to `ironic.conf`.

This applies to any changes to any `default_*_interface` options or the order of interfaces in the for the `enabled_*_interfaces` options.

Once the conductor has been restarted with the updated configuration, you should now be able to update the interface using the `baremetal node set` command. In this example we use the `network_interface` as this is most commonly where it is encountered:

```
$ baremetal node set $NAME_OR_UUID --network-interface flat
```

Note: There are additional paths one can take to remedy this sort of issue, however we encourage operators to be mindful of operational consistency when making major configuration changes.

Once you have updated the saved interfaces, you should be able to safely return the `ironic.conf` configuration change in changing what interfaces are enabled by the conductor.

Im getting Out of Memory errors

This issue, also known as the the OOMKiller got my conductor case, is where your OS system memory reaches a point where the operating system engages measures to shed active memory consumption in order to prevent a complete failure of the machine. Unfortunately this can cause unpredictable behavior.

How did I get here?

One of the major consumers of memory in a host running an ironic-conductor is transformation of disk images using the `qemu-img` tool. This tool, because the disk images it works with are both compressed and out of linear block order, requires a considerable amount of memory to efficently re-assemble and write-out a disk to a device, or to simply convert the format such as to a `raw` image.

By default, ironics configuration limits this conversion to 1 GB of RAM for the process, but each conversion does cause additional buffer memory to be used, which increases overall system memory pressure. Generally memory pressure alone from buffers will not cause an out of memory condition, but the multiple conversions or deployments running at the same time CAN cause extreme memory pressure and risk the system running out of memory.

How do I resolve this?

This can be addressed a few different ways:

- Use raw images, however these images can be substantially larger and require more data to be transmitted over the wire.
- Add more physical memory.
- Add swap space.
- Reduce concurrency, possibly via another conductor or changing the nova-compute.conf `max_concurrent_builds` parameter.
- Or finally, adjust the `[DEFAULT]minimum_required_memory` parameter in your ironic.conf file. The default should be considered a default of last resort and you may need to reserve additional memory. You may also wish to adjust the `[DEFAULT]minimum_memory_wait_retries` and `[DEFAULT]minimum_memory_wait_time` parameters.

Why does API return Node is locked by host?

This error usually manifests as HTTP error 409 on the client side:

> Node d7e2aed8-50a9-4427-baaa-f8f595e2ceb3 is locked by host 192.168.122.1, please retry after the current operation is completed.

It happens, because an operation that modifies a node is requested, while another such operation is running. The conflicting operation may be user requested (e.g. a provisioning action) or related to the internal processes (e.g. changing power state during *Power Synchronization*). The reported host name corresponds to the conductor instance that holds the lock.

Normally, these errors are transient and safe to retry after a few seconds. If the lock is held for significant time, these are the steps you can take.

First of all, check the current `provision_state` of the node:

`verifying` means that the conductor is trying to access the nodes BMC. If it happens for minutes, it means that the BMC is either unreachable or misbehaving. Double-check the information in `driver_info`, especially the BMC address and credentials.

If the access details seem correct, try resetting the BMC using, for example, its web UI.

`deploying/inspecting/cleaning` means that the conductor is doing some active work. It may include downloading or converting images, executing synchronous out-of-band deploy or clean steps, etc. A node can stay in this state for minutes, depending on various factors. Consult the conductor logs.

`available/manageable/wait call-back/clean wait` means that some background process is holding the lock. Most commonly its the power synchronization loop. Similarly to the `verifying` state, it may mean that the BMC access is broken or too slow. The conductor logs will provide you insights on what is happening.

To trace the process using conductor logs:

1. Isolate the relevant log parts. Lock messages come from the `ironic.conductor.task_manager` module. You can also check the `ironic.common.states` module for any state transitions:

```
$ grep -E '(ironic.conductor.task_manager|ironic.common.states|NodeLocked)
↪' \
    conductor.log > state.log
```

2. Find the first instance of `NodeLocked`. It may look like this (stripping timestamps and request IDs here and below for readability):

```
DEBUG ironic.conductor.task_manager [-] Attempting to get exclusive lock␣
↪on node d7e2aed8-50a9-4427-baaa-f8f595e2ceb3 (for node update) __init__␣
↪/usr/lib/python3.6/site-packages/ironic/conductor/task_manager.py:233
DEBUG ironic_lib.json_rpc.server [-] RPC error NodeLocked: Node d7e2aed8-
↪50a9-4427-baaa-f8f595e2ceb3 is locked by host 192.168.57.53, please␣
↪retry after the current operation is completed. _handle_error /usr/lib/
↪python3.6/site-packages/ironic_lib/json_rpc/server.py:179
```

   The events right before this failure will provide you a clue on why the lock is held.

3. Find the last successful **exclusive** locking event before the failure, for example:

```
DEBUG ironic.conductor.task_manager [-] Attempting to get exclusive lock␣
↪on node d7e2aed8-50a9-4427-baaa-f8f595e2ceb3 (for provision action␣
↪manage) __init__ /usr/lib/python3.6/site-packages/ironic/conductor/task_
↪manager.py:233
DEBUG ironic.conductor.task_manager [-] Node d7e2aed8-50a9-4427-baaa-
↪f8f595e2ceb3 successfully reserved for provision action manage (took 0.
↪01 seconds) reserve_node /usr/lib/python3.6/site-packages/ironic/
↪conductor/task_manager.py:350
DEBUG ironic.common.states [-] Exiting old state 'enroll' in response to␣
↪event 'manage' on_exit /usr/lib/python3.6/site-packages/ironic/common/
↪states.py:307
DEBUG ironic.common.states [-] Entering new state 'verifying' in response␣
↪to event 'manage' on_enter /usr/lib/python3.6/site-packages/ironic/
↪common/states.py:313
```

   This is your root cause, the lock is held because of the BMC credentials verification.

4. Find when the lock is released (if at all). The messages look like this:

```
DEBUG ironic.conductor.task_manager [-] Successfully released exclusive␣
↪lock for provision action manage on node d7e2aed8-50a9-4427-baaa-
↪f8f595e2ceb3 (lock was held 60.02 sec) release_resources /usr/lib/
↪python3.6/site-packages/ironic/conductor/task_manager.py:447
```

   The message tells you the reason the lock was held (`for provision action manage`) and the amount of time it was held (60.02 seconds, which is way too much for accessing a BMC).

Unfortunately, due to the way the conductor is designed, it is not possible to gracefully break a stuck lock held in `*-ing` states. As the last resort, you may need to restart the affected conductor. See *Why are my nodes stuck in a -ing state?*.

4.2.18 Power Synchronization

Baremetal Power Sync

Each Baremetal conductor process runs a periodic task which synchronizes the power state of the nodes between its database and the actual hardware. If the value of the *conductor.force_power_state_during_sync* option is set to `true` the power state in the database will be forced on the hardware and if it is set to `false` the hardware state will be forced on the database. If this periodic task is enabled, it runs at an interval defined by the *conductor.sync_power_state_interval* config option for those nodes which are not in maintenance. The requests sent to Baseboard Management Controllers (BMCs) are done with a parallelism controlled by *conductor.sync_power_state_workers*. The motivation to send out requests to BMCs in parallel is to handle misbehaving BMCs which may delay or even block the synchronization otherwise.

Note: In deployments with many nodes and IPMI as the configured BMC protocol, the default values of a 60 seconds power sync interval and 8 worker threads may lead to a high rate of required retries due to client-side UDP packet loss (visible via the corresponding warnings in the conductor logs). While Ironic automatically retries to get the power status for the affected nodes, the failure rate may be reduced by increasing the power sync cycle, e.g. to 300 seconds, and/or by reducing the number of power sync workers, e.g. to 2. Pleae keep in mind, however, that depending on the concrete setup increasing the power sync interval may have an impact on other components relying on up-to-date power states.

Compute-Baremetal Power Sync

Each `nova-compute` process in the Compute service runs a periodic task which synchronizes the power state of servers between its database and the compute driver. If enabled, it runs at an interval defined by the *sync_power_state_interval* config option on the `nova-compute` process. In case of the compute driver being baremetal driver, this sync will happen between the databases of the compute and baremetal services. Since the sync happens on the `nova-compute` process, the state in the compute database will be forced on the baremetal database in case of inconsistencies. Hence a node which was put down using the compute service API cannot be brought up through the baremetal service API since the power sync task will regard the compute services knowledge of the power state as the source of truth. In order to get around this disadvantage of the compute-baremetal power sync, baremetal service does power state change callbacks to the compute service using external events.

Power State Change Callbacks to the Compute Service

Whenever the Baremetal service changes the power state of a node, it can issue a notification to the Compute service. The Compute service will consume this notification and update the power state of the instance in its database. By conveying all the power state changes to the compute service, the baremetal service becomes the source of truth thus preventing the compute service from forcing wrong power states on the physical instance during the compute-baremetal power sync. It also adds the possibility of bringing up/down a physical instance through the baremetal service API even if it was put down/up through the compute service API.

This change requires the *nova* section and the necessary authentication options like the *nova.auth_url* to be defined in the configuration file of the baremetal service. If it is not configured the baremetal service will not be able to send notifications to the compute service and it will fall back to the behaviour of the

compute service forcing power states on the baremetal service during the power sync. See *nova* group for more details on the available config options.

In case of baremetal stand alone deployments where there is no compute service running, the *nova.send_power_notifications* config option should be set to `False` to disable power state change callbacks to the compute service.

Note: The baremetal service sends notifications to the compute service only if the target power state is `power on` or `power off`. Other error and `None` states will be ignored. In situations where the power state change is originally coming from the compute service, the notification will still be sent by the baremetal service and it will be a no-op on the compute service side with a debug log stating the node is already powering on/off.

Note: Although an exclusive lock is used when sending notifications to the compute service, there can still be a race condition if the compute-baremetal power sync happens to happen a nano-second before the power state change event is received from the baremetal service in which case the power state from compute services database will be forced on the node.

Power fault and recovery

When *Baremetal Power Sync* is enabled, and the Bare Metal service loses access to a node (usually because of invalid credentials, BMC issues or networking interruptions), the node enters `maintenance` mode and its `fault` field is set to `power failure`. The exact reason is stored in the `maintenance_reason` field.

As always with maintenance mode, only a subset of operations will work on such nodes, and both the Compute service and the Ironics native allocation API will refuse to pick them. Any in-progress operations will either pause or fail.

The conductor responsible for the node will try to recover the connection periodically (with the interval configured by the *conductor.power_failure_recovery_interval* option). If the power sync is successful, the `fault` field is unset and the node leaves the maintenance mode.

Note: This only applies to automatic maintenance mode with the `fault` field set. Maintenance mode set manually is never left automatically.

Alternatively, you can disable maintenance mode yourself once the problem is resolved:

```
baremetal node maintenance unset <IRONIC NODE>
```

4.2.19 Node Multi-Tenancy

This guide explains the steps needed to enable node multi-tenancy. This feature enables non-admins to perform API actions on nodes, limited by policy configuration. The Bare Metal service supports two kinds of non-admin users:

- Owner: owns specific nodes and performs administrative actions on them
- Lessee: receives temporary and limited access to a node

Setting the Owner and Lessee

Non-administrative access to a node is controlled through a nodes `owner` or `lessee` attribute:

```
baremetal node set --owner 080925ee2f464a2c9dce91ee6ea354e2  node-7
baremetal node set --lessee 2a210e5ff114c8f2b6e994218f51a904  node-10
```

Configuring the Bare Metal Service Policy

By default, the Bare Metal service policy is configured so that a node owner or lessee has no access to any node APIs. However, the policy *policy file* contains rules that can be used to enable node API access:

```
# Owner of node
#"is_node_owner": "project_id:%(node.owner)s"

# Lessee of node
#"is_node_lessee": "project_id:%(node.lessee)s"
```

An administrator can then modify the policy file to expose individual node APIs as follows:

```
# Change Node provision status
# PUT  /nodes/{node_ident}/states/provision
#"baremetal:node:set_provision_state": "rule:is_admin"
"baremetal:node:set_provision_state": "rule:is_admin or rule:is_node_owner or␣
↪rule:is_node_lessee"

# Update Node records
# PATCH  /nodes/{node_ident}
#"baremetal:node:update": "rule:is_admin or rule:is_node_owner"
```

In addition, it is safe to expose the `baremetal:node:list` rule, as the node list function now filters non-admins by owner and lessee:

```
# Retrieve multiple Node records, filtered by owner
# GET  /nodes
# GET  /nodes/detail
#"baremetal:node:list": "rule:baremetal:node:get"
"baremetal:node:list": ""
```

Note that `baremetal:node:list_all` permits users to see all nodes regardless of owner/lessee, so it should remain restricted to admins.

Ports

Port APIs can be similarly exposed to node owners and lessees:

```
# Retrieve Port records
# GET  /ports/{port_id}
# GET  /nodes/{node_ident}/ports
# GET  /nodes/{node_ident}/ports/detail
# GET  /portgroups/{portgroup_ident}/ports
# GET  /portgroups/{portgroup_ident}/ports/detail
#"baremetal:port:get": "rule:is_admin or rule:is_observer"
"baremetal:port:get": "rule:is_admin or rule:is_observer or rule:is_node_
↪owner or rule:is_node_lessee"

# Retrieve multiple Port records, filtered by owner
# GET  /ports
# GET  /ports/detail
#"baremetal:port:list": "rule:baremetal:port:get"
"baremetal:port:list": ""
```

Allocations

Allocations respect node tenancy as well. A restricted allocation creates an allocation tied to a project, and that can only match nodes where that project is the owner or lessee. Here is a sample set of allocation policy rules that allow non-admins to use allocations effectively:

```
# Retrieve Allocation records
# GET  /allocations/{allocation_id}
# GET  /nodes/{node_ident}/allocation
#"baremetal:allocation:get": "rule:is_admin or rule:is_observer"
"baremetal:allocation:get": "rule:is_admin or rule:is_observer or rule:is_
↪allocation_owner"

# Retrieve multiple Allocation records, filtered by owner
# GET  /allocations
#"baremetal:allocation:list": "rule:baremetal:allocation:get"
"baremetal:allocation:list": ""

# Retrieve multiple Allocation records
# GET  /allocations
#"baremetal:allocation:list_all": "rule:baremetal:allocation:get"

# Create Allocation records
# POST  /allocations
#"baremetal:allocation:create": "rule:is_admin"

# Create Allocation records that are restricted to an owner
# POST  /allocations
#"baremetal:allocation:create_restricted": "rule:baremetal:allocation:create"
```

(continues on next page)

(continued from previous page)

```
"baremetal:allocation:create_restricted": ""

# Delete Allocation records
# DELETE  /allocations/{allocation_id}
# DELETE  /nodes/{node_ident}/allocation
#"baremetal:allocation:delete": "rule:is_admin"
"baremetal:allocation:delete": "rule:is_admin or rule:is_allocation_owner"

# Change name and extra fields of an allocation
# PATCH  /allocations/{allocation_id}
#"baremetal:allocation:update": "rule:is_admin"
"baremetal:allocation:update": "rule:is_admin or rule:is_allocation_owner"
```

Deployment and Metalsmith

Provisioning a node requires a specific set of APIs to be made available. The following policy specifications are enough to allow a node owner to use Metalsmith to deploy upon a node:

```
"baremetal:node:get": "rule:is_admin or rule:is_observer or rule:is_node_owner
↪"
"baremetal:node:list": ""
"baremetal:node:update_extra": "rule:is_admin or rule:is_node_owner"
"baremetal:node:update_instance_info": "rule:is_admin or rule:is_node_owner"
"baremetal:node:validate": "rule:is_admin or rule:is_node_owner"
"baremetal:node:set_provision_state": "rule:is_admin or rule:is_node_owner"
"baremetal:node:vif:list": "rule:is_admin or rule:is_node_owner"
"baremetal:node:vif:attach": "rule:is_admin or rule:is_node_owner"
"baremetal:node:vif:detach": "rule:is_admin or rule:is_node_owner"
"baremetal:allocation:get": "rule:is_admin or rule:is_observer or rule:is_
↪allocation_owner"
"baremetal:allocation:list": ""
"baremetal:allocation:create_restricted": ""
"baremetal:allocation:delete": "rule:is_admin or rule:is_allocation_owner"
"baremetal:allocation:update": "rule:is_admin or rule:is_allocation_owner"
```

4.2.20 Fast-Track Deployment

Fast track is a mode of operation where the Bare Metal service keeps a machine powered on with the agent running between provisioning operations. It is first booted during in-band inspection or cleaning (whatever happens first) and is only shut down before rebooting into the final instance. Depending on the configuration, this mode can save several reboots and is particularly useful for scenarios where nodes are enrolled, prepared and provisioned within a short period of time.

> **Warning:** Fast track deployment targets standalone use cases and is only tested with the noop networking. The case where inspection, cleaning and provisioning networks are different is not supported.

Enabling

Fast track is off by default and should be enabled in the configuration:

```
[deploy]
fast_track = true
```

Starting with the Yoga release series, it can also be enabled or disabled per node:

```
baremetal node set <node> --driver-info fast_track=true
```

Inspection

If using *In-band inspection*, you need to tell ironic-inspector not to power off nodes afterwards. Depending on the inspection mode (managed or unmanaged), you need to configure two places. In `ironic.conf`:

```
[inspector]
power_off = false
```

And in `inspector.conf`:

```
[processing]
power_off = false
```

Finally, you need to update the inspection PXE configuration to include the `ipa-api-url` kernel parameter, pointing at the **ironic** endpoint, in addition to the existing `ipa-inspection-callback-url`.

4.2.21 Booting a Ramdisk or an ISO

Ironic supports booting a user provided ramdisk or an ISO image (starting with the Victoria release) instead of deploying a node. Most commonly this is performed when an instance is booted via PXE, iPXE or Virtual Media, with the only local storage contents being those in memory. It is suported by `pxe`, `ipxe`, `redfish-virtual-media` and `ilo-virtual-media` boot interfaces.

Configuration

Ramdisk/ISO boot requires using the `ramdisk` deploy interface. As with most non-default interfaces, it must be enabled and set for a node to be utilized:

```
[DEFAULT]
...
enabled_deploy_interfaces = direct,ramdisk
...
```

Once enabled and the conductor(s) have been restarted, the interface can be set upon creation of a new node:

```
baremetal node create --driver ipmi \
    --deploy-interface ramdisk \
    --boot-interface ipxe
```

or update an existing node:

```
baremetal node set <NODE> --deploy-interface ramdisk
```

You can also use it with *redfish virtual media* instead of iPXE.

Creating a ramdisk

A ramdisk can be created using the `ironic-ramdisk-base` element from ironic-python-agent-builder, e.g. with Debian:

```
export ELEMENTS_PATH=/opt/stack/ironic-python-agent-builder/dib
disk-image-create -o /output/ramdisk \
    debian-minimal ironic-ramdisk-base openssh-server dhcp-all-interfaces
```

You should consider using the following elements:

- openssh-server to install the SSH server since its not provided by default by some minimal images.
- devuser or dynamic-login to provide SSH access.
- dhcp-all-interfaces or simple-init to configure networking.

The resulting files (`/output/ramdisk.kernel` and `/output/ramdisk.initramfs` in this case) can then be used when *Booting a ramdisk*.

Booting a ramdisk

Pass the kernel and ramdisk as normally, also providing the ramdisk as an image source, for example,

```
baremetal node set <NODE> \
    --instance-info kernel=http://path/to/ramdisk.kernel \
    --instance-info ramdisk=http://path/to/ramdisk.initramfs
baremetal node deploy <NODE>
```

Note: Before the Xena release, the `image_source` field was also required:

```
--instance-info image_source=http://path/to/ramdisk.initramfs
```

Booting an ISO

The `ramdisk` deploy interface can also be used to boot an ISO image. For example,

```
baremetal node set <NODE> \
    --instance-info boot_iso=http://path/to/boot.iso
baremetal node deploy <NODE>
```

Warning: This feature, when utilized with the `ipxe` `boot_interface`, will only allow a kernel and ramdisk to be booted from the supplied ISO file. Any additional contents, such as additional ramdisk contents or installer package files will be unavailable after the boot of the Operating System. Operators wishing to leverage this functionality for actions such as OS installation should explore use of the standard `ramdisk` `deploy_interface` along with the `instance_info/kernel_append_params` setting to pass arbitrary settings such as a mirror URL for the initial ramdisk to load data from. This is a limitation of iPXE and the overall boot process of the operating system where memory allocated by iPXE is released.

By default the Bare Metal service will cache the ISO locally and serve from its HTTP server. If you want to avoid that, set the following:

```
baremetal node set <NODE> \
    --instance-info ramdisk_image_download_source=http
```

ISO images are also cached across deployments, similarly to how it is done for normal instance images. The URL together with the last modified response header are used to determine if an image needs updating.

Limitations

The intended use case is for advanced scientific and ephemeral workloads where the step of writing an image to the local storage is not required or desired. As such, this interface does come with several caveats:

- Configuration drives are not supported with network boot, only with Redfish virtual media.
- Disk image contents are not written to the bare metal node.
- Users and Operators who intend to leverage this interface should expect to leverage a metadata service, custom ramdisk images, or the `instance_info/ramdisk_kernel_arguments` parameter to add options to the kernel boot command line.
- When using PXE/iPXE boot, bare metal nodes must continue to have network access to PXE and iPXE network resources. This is contrary to most tenant networking enabled configurations where this access is restricted to the provisioning and cleaning networks
- As with all deployment interfaces, automatic cleaning of the node will still occur with the contents of any local storage being wiped between deployments.

Common options

Disable persistent boot device for ramdisk iso boot

For iso boot, Ironic sets the boot target to continuously boot from the iso attached over virtual media. This behaviour may not always be desired e.g. if the vmedia is installing to hard drive and then rebooting. In order to instead set the virtual media to be one time boot Ironic provides the `force_persistent_boot_device` flag in the nodes `driver_info`. Which can be set to `Never`:

```
$ openstack baremetal node set --driver-info force_persistent_boot_device=
↪'Never' <node>
```

4.2.22 Hardware Burn-in

Overview

Workflows to onboard new hardware often include a stress-testing step to provoke early failures and to avoid that these load-triggered issues only occur when the nodes have already moved to production. These `burn-in` tests typically include CPU, memory, disk, and network. With the Xena release, Ironic supports such tests as part of the cleaning framework.

The burn-in steps rely on standard tools such as stress-ng for CPU and memory, or fio for disk and network. The burn-in cleaning steps are part of the generic hardware manager in the Ironic Python Agent (IPA) and therefore the agent ramdisk does not need to be bundled with a specific IPA hardware manager to have them available.

Each burn-in step accepts (or in the case of network: needs) some basic configuration options, mostly to limit the duration of the test and to specify the amount of resources to be used. The options are set on a nodes `driver-info` and prefixed with `agent_burnin_`. The options available for the individual tests will be outlined below.

CPU burn-in

The options, following a *agent_burnin_* + stress-ng stressor (*cpu*) + stress-ng option schema, are:

- `agent_burnin_cpu_timeout` (default: 24 hours)
- `agent_burnin_cpu_cpu` (default: 0, meaning all CPUs)

to limit the overall runtime and to pick the number of CPUs to stress.

For instance, in order to limit the time of the CPU burn-in to 10 minutes do:

```
baremetal node set --driver-info agent_burnin_cpu_timeout=600 \
    $NODE_NAME_OR_UUID
```

Then launch the test with:

```
baremetal node clean --clean-steps '[{"step": "burnin_cpu", \
    "interface": "deploy"}]' $NODE_NAME_OR_UUID
```

Memory burn-in

The options, following a *agent_burnin_* + stress-ng stressor (*vm*) + stress-ng option schema, are:

- `agent_burnin_vm_timeout` (default: 24 hours)
- `agent_burnin_vm_vm-bytes` (default: 98%)

to limit the overall runtime and to set the fraction of RAM to stress.

For instance, in order to limit the time of the memory burn-in to 1 hour and the amount of RAM to be used to 75% run:

```
baremetal node set --driver-info agent_burnin_vm_timeout=3600 \
    $NODE_NAME_OR_UUID
baremetal node set --driver-info agent_burnin_vm_vm-bytes=75% \
    $NODE_NAME_OR_UUID
```

Then launch the test with:

```
baremetal node clean --clean-steps '[{"step": "burnin_memory", \
    "interface": "deploy"}]' $NODE_NAME_OR_UUID
```

Disk burn-in

The options, following a *agent_burnin_* + fio stressor (*fio_disk*) + fio option schema, are:

- agent_burnin_fio_disk_runtime (default: 0, meaning no time limit)
- agent_burnin_fio_disk_loops (default: 4)

to set the time limit and the number of iterations when going over the disks.

For instance, in order to limit the number of loops to 2 set:

```
baremetal node set --driver-info agent_burnin_fio_disk_loops=2 \
    $NODE_NAME_OR_UUID
```

Then launch the test with:

```
baremetal node clean --clean-steps '[{"step": "burnin_disk", \
    "interface": "deploy"}]' $NODE_NAME_OR_UUID
```

Network burn-in

Burning in the network needs a little more config, since we need a pair of nodes to perform the test. Therefore, this test needs to set `agent_burnin_fio_network_config` JSON which requires a `role` field (values: `reader`, `writer`) and a `partner` field (value is the hostname of the other node to test), like:

```
baremetal node set --driver-info agent_burnin_fio_network_config= \
    '{"role": "writer", "partner": "$HOST2"}' $NODE_NAME_OR_UUID1
baremetal node set --driver-info agent_burnin_fio_network_config= \
    '{"role": "reader", "partner": "$HOST1"}' $NODE_NAME_OR_UUID2
```

In addition and similar to the other tests, there is a runtime option to be set (only on the writer):

```
baremetal node set --driver-info agent_burnin_fio_network_runtime=600 \
    $NODE_NAME_OR_UUID
```

Then launch the test with:

```
baremetal node clean --clean-steps '[{"step": "burnin_network",\
    "interface": "deploy"}]' $NODE_NAME_OR_UUID1
baremetal node clean --clean-steps '[{"step": "burnin_network",\
    "interface": "deploy"}]' $NODE_NAME_OR_UUID2
```

Both nodes will wait for the other node to show up and block while waiting. If the partner does not show up, the cleaning timeout will step in.

Logging

Since most of the burn-in steps are also providing information about the performance of the stressed components, keeping this information for verification or acceptance purposes may be desirable. By default, the output of the burn-in tools goes to the journal of the Ironic Python Agent and is therefore sent back as an archive to the conductor. In order to consume the output of the burn-in steps more easily, or even in real-time, the nodes can be configured to store the output of the individual steps to files in the ramdisk (from where they can be picked up by a logging pipeline).

The configuration of the outpout file is done via one of `agent_burnin_cpu_outputfile`, `agent_burnin_vm_outputfile`, `agent_burnin_fio_disk_outputfile`, and `agent_burnin_fio_network_outputfile` parameters which need to be added to a node like:

```
baremetal node set --driver-info agent_burnin_cpu_outputfile=\
    '/var/log/burnin.cpu' $NODE_NAME_OR_UUID
```

Additional Information

All tests can be aborted at any moment with

```
baremetal node abort $NODE_NAME_OR_UUID
```

One can also launch multiple tests which will be run in sequence, e.g.:

```
baremetal node clean --clean-steps '[{"step": "burnin_cpu",\
   "interface": "deploy"}, {"step": "burnin_memory",\
   "interface": "deploy"}]' $NODE_NAME_OR_UUID
```

If desired, configuring `fast-track` may be helpful here as it allows to keep the node up between consecutive calls of `baremetal node clean`.

4.2.23 Vendor Passthru

The bare metal service allows drivers to expose vendor-specific API known as *vendor passthru*.

Node Vendor Passthru

Drivers may implement a passthrough API, which is accessible via the `/v1/nodes/<Node UUID or Name>/vendor_passthru?method={METHOD}` endpoint. Beyond basic checking, Ironic does not introspect the message body and simply passes it through to the relevant driver.

A method:

- can support one or more HTTP methods (for example, GET, POST)
- is asynchronous or synchronous
  - For asynchronous methods, a 202 (Accepted) HTTP status code is returned to indicate that the request was received, accepted and is being acted upon. No body is returned in the response.
  - For synchronous methods, a 200 (OK) HTTP status code is returned to indicate that the request was fulfilled. The response may include a body.
- can require an exclusive lock on the node. This only occurs if the method doesnt specify require_exclusive_lock=False in the decorator. If an exclusive lock is held on the node, other requests for the node will be delayed and may fail with an HTTP 409 (Conflict) error code.

This endpoint exposes a nodes driver directly, and as such, it is expressly not part of Ironics standard REST API. There is only a single HTTP endpoint exposed, and the semantics of the message body are determined solely by the driver. Ironic makes no guarantees about backwards compatibility; this is solely up to the discretion of each drivers author.

To get information about all the methods available via the vendor_passthru endpoint for a particular node, use CLI:

```
$ baremetal node passthru list <redfish-node>
+----------------------+----------------------+-------+--------------------
↪--------------------------------------------------------------------------
↪--------------------------------------------------------------------------
↪---------------------------+-------------------------+
| Name                      | Supported HTTP methods | Async | Description
↪                                                                          
↪                                                                          
↪                           | Response is attachment |
+----------------------+----------------------+-------+--------------------
↪--------------------------------------------------------------------------
↪--------------------------------------------------------------------------
↪---------------------------+-------------------------+
| create_subscription   | POST                   | False | Creates a
↪subscription on a node. Required argument: a dictionary of {'destination':
↪'destination_url'}
↪                                       | False                  |
| delete_subscription   | DELETE                 | False | Delete a
↪subscription on a node. Required argument: a dictionary of {'id':
↪'subscription_bmc_id'}
↪                                                | False                  |
```

(continues on next page)

(continued from previous page)

```
| eject_vmedia          | POST                    | True  | Eject a virtual␣
↪media device. If no device is provided then all attached devices will be␣
↪ejected. Optional arguments: 'boot_device' - the boot device to eject,␣
↪either 'cd', 'dvd', 'usb', or 'floppy' | False                  |
| get_all_subscriptions | GET                     | False | Returns all␣
↪subscriptions on the node.                                              ␣
↪                                                                        ␣
↪                                  | False                 |
| get_subscription      | GET                     | False | Get a subscription␣
↪on the node. Required argument: a dictionary of {'id': 'subscription_bmc_id
↪'}                                                                      ␣
↪                              | False                 |
+----------------------+-------------------------+-------+-------------------
↪-------------------------------------------------------------------------
↪-------------------------------------------------------------------------
↪---------------------------+------------------------+
```

The response will contain information for each method, such as the methods name, a description, the HTTP methods supported, and whether its asynchronous or synchronous.

You can call a method with CLI, for example:

```
$ baremetal node passthru call <redfish-node> eject_vmedia
```

Driver Vendor Passthru

Drivers may implement an API for requests not related to any node, at `/v1/drivers/<driver name>/vendor_passthru?method={METHOD}`.

A method:

- can support one or more HTTP methods (for example, GET, POST)
- is asynchronous or synchronous
  - For asynchronous methods, a 202 (Accepted) HTTP status code is returned to indicate that the request was received, accepted and is being acted upon. No body is returned in the response.
  - For synchronous methods, a 200 (OK) HTTP status code is returned to indicate that the request was fulfilled. The response may include a body.

Note: Unlike methods in *Node Vendor Passthru*, a request does not lock any resource, so it will not delay other requests and will not fail with an HTTP 409 (Conflict) error code.

Ironic makes no guarantees about the semantics of the message BODY sent to this endpoint. That is left up to each drivers author.

To get information about all the methods available via the driver vendor_passthru endpoint, use CLI:

```
$ baremetal driver passthru list redfish
```

The response will contain information for each method, such as the methods name, a description, the HTTP methods supported, and whether its asynchronous or synchronous.

Warning: Currently only the methods available in the default interfaces of the hardware type are available.

You can call a method with CLI, for example:

```
$ baremetal driver passthru call <driver> <method>
```

4.2.24 Drivers, Hardware Types and Hardware Interfaces

4.2.25 Advanced Topics

Ceph Object Gateway support

Overview

Ceph project is a powerful distributed storage system. It contains object store and provides a RADOS Gateway Swift API which is compatible with OpenStack Swift API.

Ironic added support for RADOS Gateway temporary URL in the Mitaka release.

Configure Ironic and Glance with RADOS Gateway

1. Install Ceph storage with RADOS Gateway. See Ceph documentation.
2. Configure RADOS Gateway to use keystone for authentication. See Integrating with OpenStack Keystone
3. Register RADOS Gateway endpoint in the keystone catalog, with the same format swift uses, as the `object-store` service. URL example:

   `http://rados.example.com:8080/swift/v1/AUTH_$(project_id)s`.

   In the ceph configuration, make sure radosgw is configured with the following value:

   ```
   rgw swift account in url = True
   ```

4. Configure Glance API service for RADOS Swift API as backend. Edit the configuration file for the Glance API service (is typically located at `/etc/glance/glance-api.conf`):

   ```
   [glance_store]

   stores = file, http, swift
   default_store = swift
   default_swift_reference=ref1
   swift_store_config_file=/etc/glance/glance-swift-creds.conf
   swift_store_container = glance
   swift_store_create_container_on_put = True
   ```

In the file referenced in `swift_store_config_file` option, add the following:

```
[ref1]
user = <service project>:<service user name>
key = <service user password>
user_domain_id = default
project_domain_id = default
auth_version = 3
auth_address = http://keystone.example.com/identity
```

Values for user and key options correspond to keystone credentials for RADOS Gateway service user.

Note: RADOS Gateway uses FastCGI protocol for interacting with HTTP server. Read your HTTP server documentation if you want to enable HTTPS support.

5. Restart Glance API service and upload all needed images.
6. If youre using custom container name in RADOS, change Ironic configuration file on the conductor host(s) as follows:

```
[glance]

swift_container = glance
```

7. Restart Ironic conductor service(s).

Building images for Windows

We can use `New-WindowsOnlineImage` in windows-openstack-imaging-tools tool as an option to create Windows images (whole disk images) corresponding boot modes which will support for Windows NIC Teaming. And allow the utilization of link aggregation when the instance is spawned on hardware servers (Bare metals).

Requirements:

- A Microsoft Windows Server Operating System along with `Hyper-V virtualization` enabled, `PowerShell` version >=4 supported, `Windows Assessment and Deployment Kit`, in short `Windows ADK`.
- The windows Server compatible drivers.
- Working git environment.

Preparation:

- Download a Windows Server 2012R2/ 2016 installation ISO.
- Install Windows Server 2012R2/ 2016 OS on workstation PC along with following feature:
  - Enable Hyper-V virtualization.
  - Install PowerShell 4.0.
  - Install Git environment & import git proxy (if have).
  - Create new `Path` in Microsoft Windows Server Operating System which support for sub-module update via `git submodule update init` command:

```
- Variable name: Path
- Variable value: C:\Windows\System32\WindowsPowerShell\v1.0\;C:\
↪Program Files\Git\bin
```

  - Rename virtual switch name in Windows Server 2012R2/ 2016 in `Virtual Switch Manager` into *external*.

Implementation:

- `Step 1`: Create folders: `C:\<folder_name_1>` where output images will be located, `C:\<folder_name_2>` where you need to place the necessary hardware drivers.
- `Step 2`: Copy and extract necessary hardware drivers in `C:\<folder_name_2>`.
- `Step 3`: Insert or burn Windows Server 2016 ISO to `D:\`.
- `Step 4`: Download `windows-openstack-imaging-tools` tools.

```
git clone https://github.com/cloudbase/windows-openstack-imaging-tools.git
```

- `Step 5`: Create & running script *create-windows-cloud-image.ps1*:

```
git submodule update --init
Import-Module WinImageBuilder.psm1
$windowsImagePath = "C:\<folder_name_1>\<output_file_name>.qcow2"
$VirtIOISOPath = "C:\<folder_name_1>\virtio.iso"
$virtIODownloadLink = "https://fedorapeople.org/groups/virt/virtio-win/
↪direct-downloads/archive-virtio/virtio-win-0.1.133-2/virtio-win.iso"
(New-Object System.Net.WebClient).DownloadFile($virtIODownloadLink,
↪$VirtIOISOPath)
$wimFilePath = "D:\sources\install.wim"
$extraDriversPath = "C:\<folder_name_2>\"
$image = (Get-WimFileImagesInfo -WimFilePath $wimFilePath)[1]
$switchName = 'external'
New-WindowsOnlineImage -WimFilePath $wimFilePath
  -ImageName $image.ImageName ` -WindowsImagePath $windowsImagePath -Type
↪'KVM' -ExtraFeatures @() `
  -SizeBytes 20GB -CpuCores 2 -Memory 2GB -SwitchName $switchName ` -
↪ProductKey $productKey -DiskLayout 'BIOS' `
```

(continues on next page)

(continued from previous page)

```
  -ExtraDriversPath $extraDriversPath ` -InstallUpdates:$false -
↪AdministratorPassword 'Pa$$w0rd' `
  -PurgeUpdates:$true -DisableSwap:$true
```

After executing this command you will get two output files, first one being C:<folder_name_1><output_file_name>.qcow2, which is the resulting windows whole disk image and C:<folder_name_1>virtio.iso, which is virtio iso contains all the synthetic drivers for the KVM hypervisor.

See example_windows_images for more details and examples.

Note: We can change `SizeBytes`, `CpuCores` and `Memory` depending on requirements.

Emitting Software Metrics

Beginning with the Newton (6.1.0) release, the ironic services support emitting internal performance data to statsd. This allows operators to graph and understand performance bottlenecks in their system.

This guide assumes you have a statsd server setup. For information on using and configuring statsd, please see the statsd README and documentation.

These performance measurements, herein referred to as metrics, can be emitted from the Bare Metal service, including ironic-api, ironic-conductor, and ironic-python-agent. By default, none of the services will emit metrics.

Configuring the Bare Metal Service to Enable Metrics

Enabling metrics in ironic-api and ironic-conductor

The ironic-api and ironic-conductor services can be configured to emit metrics to statsd by adding the following to the ironic configuration file, usually located at `/etc/ironic/ironic.conf`:

```
[metrics]
backend = statsd
```

If a statsd daemon is installed and configured on every host running an ironic service, listening on the default UDP port (8125), no further configuration is needed. If you are using a remote statsd server, you must also supply connection information in the ironic configuration file:

```
[metrics_statsd]
# Point this at your environments' statsd host
statsd_host = 192.0.2.1
statsd_port = 8125
```

Enabling metrics in ironic-python-agent

The ironic-python-agent process receives its configuration in the response from the initial lookup request to the ironic-api service. This means to configure ironic-python-agent to emit metrics, you must enable the agent metrics backend in your ironic configuration file on all ironic-conductor hosts:

```
[metrics]
agent_backend = statsd
```

In order to reliably emit metrics from the ironic-python-agent, you must provide a statsd server that is reachable from both the configured provisioning and cleaning networks. The agent statsd connection information is configured in the ironic configuration file as well:

```
[metrics_statsd]
# Point this at a statsd host reachable from the provisioning and cleaning␣
↪nets
agent_statsd_host = 198.51.100.2
agent_statsd_port = 8125
```

Types of Metrics Emitted

The Bare Metal service emits timing metrics for every API method, as well as for most driver methods. These metrics measure how long a given method takes to execute.

A deployer with metrics enabled should expect between 100 and 500 distinctly named data points to be emitted from the Bare Metal service. This will increase if the metrics.preserve_host option is set to true or if multiple drivers are used in the Bare Metal deployment. This estimate may be used to determine if a deployer needs to scale their metrics backend to handle the additional load before enabling metrics. To see which metrics have changed names or have been removed between releases, refer to the ironic release notes.

Note: With the default statsd configuration, each timing metric may create additional metrics due to how statsd handles timing metrics. For more information, see statds documentation on metric types.

The ironic-python-agent ramdisk emits timing metrics for every API method.

Deployers who use custom HardwareManagers can emit custom metrics for their hardware. For more information on custom HardwareManagers, and emitting metrics from them, please see the ironic-python-agent documentation.

Adding New Metrics

If youre a developer, and would like to add additional metrics to ironic, please see the ironic-lib developer documentation for details on how to use the metrics library. A release note should also be created each time a metric is changed or removed to alert deployers of the change.

API Audit Logging

Audit middleware supports delivery of CADF audit events via Oslo messaging notifier capability. Based on *notification_driver* configuration, audit events can be routed to messaging infrastructure (notification_driver = messagingv2) or can be routed to a log file (*[oslo_messaging_notifications]/driver = log*).

Audit middleware creates two events per REST API interaction. First event has information extracted from request data and the second one has request outcome (response).

Enabling API Audit Logging

Audit middleware is available as part of *keystonemiddleware* (>= 1.6) library. For information regarding how audit middleware functions refer here.

Auditing can be enabled for the Bare Metal service by making the following changes to `/etc/ironic/ironic.conf`.

1. To enable audit logging of API requests:

```
[audit]
...
enabled=true
```

2. To customize auditing API requests, the audit middleware requires the audit_map_file setting to be defined. Update the value of configuration setting audit_map_file to set its location. Audit map file configuration options for the Bare Metal service are included in the etc/ironic/ironic_api_audit_map.conf.sample file. To understand CADF format specified in ironic_api_audit_map.conf file refer to CADF Format.:

```
[audit]
...
audit_map_file=/etc/ironic/api_audit_map.conf
```

3. Comma separated list of Ironic REST API HTTP methods to be ignored during audit. It is used only when API audit is enabled. For example:

```
[audit]
...
ignore_req_list=GET,POST
```

Sample Audit Event

Following is the sample of audit event for ironic node list request.

```
{
   "event_type":"audit.http.request",
   "timestamp":"2016-06-15 06:04:30.904397",
   "payload":{
      "typeURI":"http://schemas.dmtf.org/cloud/audit/1.0/event",
      "eventTime":"2016-06-15T06:04:30.903071+0000",
      "target":{
```

(continues on next page)

(continued from previous page)


```
            "id":"ironic",
            "typeURI":"unknown",
            "addresses":[
               {
                  "url":"http://{ironic_admin_host}:6385",
                  "name":"admin"
               },
               {
                  "url":"http://{ironic_internal_host}:6385",
                  "name":"private"
               },
               {
                  "url":"http://{ironic_public_host}:6385",
                  "name":"public"
               }
            ],
            "name":"ironic"
         },
         "observer":{
            "id":"target"
         },
         "tags":[
            "correlation_id?value=685f1abb-620e-5d5d-b74a-b4135fb32373"
         ],
         "eventType":"activity",
         "initiator":{
            "typeURI":"service/security/account/user",
            "name":"admin",
            "credential":{
               "token":"***",
               "identity_status":"Confirmed"
            },
            "host":{
               "agent":"python-ironicclient",
               "address":"10.1.200.129"
            },
            "project_id":"d8f52dd7d9e1475dbbf3ba47a4a83313",
            "id":"8c1a948bad3948929aa5d5b50627a174"
         },
         "action":"read",
         "outcome":"pending",
         "id":"061b7aa7-5879-5225-a331-c002cf23cb6c",
         "requestPath":"/v1/nodes/?associated=True"
      },
      "priority":"INFO",
      "publisher_id":"ironic-api",
      "message_id":"2f61ebaa-2d3e-4023-afba-f9fca6f21fc2"
}
```

Bare Metal Service state report (via Guru Meditation Reports)

The Bare Metal service contains a mechanism whereby developers and system administrators can generate a report about the state of running Bare Metal executables (ironic-api and ironic-conductor). This report is called a Guru Meditation Report (GMR for short). GMR provides useful debugging information that can be used to obtain an accurate view on the current live state of the system. For example, what threads are running, what configuration parameters are in effect, and more. The eventlet backdoor facility provides an interactive shell interface for any eventlet based process, allowing an administrator to telnet to a pre-defined port and execute a variety of commands.

Configuration

The GMR feature is optional and requires the oslo.reports package to be installed. For example, using pip:

```
pip install 'oslo.reports>=1.18.0'
```

Generating a GMR

A *GMR* can be generated by sending the *USR2* signal to any Bare Metal process that supports it. The *GMR* will then be output to stderr for that particular process. For example:

Suppose that `ironic-api` has process ID `6385`, and was run with `2>/var/log/ironic/ironic-api-err.log`. Then, sending the *USR* signal:

```
kill -USR2 6385
```

will trigger the Guru Meditation report to be printed to `/var/log/ironic/ironic-api-err.log`.

Structure of a GMR

The *GMR* consists of the following sections:

Package Shows information about the package to which this process belongs, including version information.

Threads Shows stack traces and thread IDs for each of the threads within this process.

Green Threads Shows stack traces for each of the green threads within this process (green threads dont have thread IDs).

Configuration Lists all the configuration options currently accessible via the CONF object for the current process.

Agent Token

Purpose

The concept of agent tokens is to provide a mechanism by which the relationship between an operating deployment of the Bare Metal Service and an instance of the `ironic-python-agent` is verified. In a sense, this token can be viewed as a session identifier or authentication token.

> **Warning:** This functionality does not remove the risk of a man-in-the-middle attack that could occur from connection intercept or when TLS is not used for all communication.

This becomes useful in the case of deploying an edge node where intermediate networks are not trustworthy.

How it works

These tokens are provided in one of two ways to the running agent.

1. A pre-generated token which is embedded into virtual media ISOs.
2. A one-time generated token that are provided upon the first lookup of the node.

In both cases, the tokens are a randomly generated using the Python `secrets` library. As of mid-2020, the default length is 43 characters.

Once the token has been provided, the token cannot be retrieved or accessed. It remains available to the conductors, and is stored in memory of the `ironic-python-agent`.

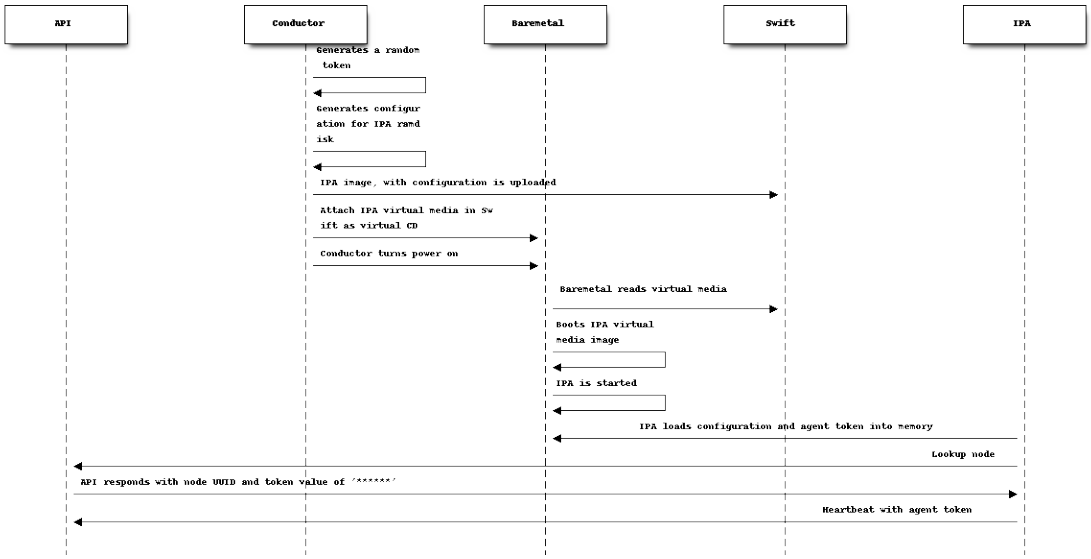
Note: In the case of the token being embedded with virtual media, it is read from a configuration file with-in the image. Ideally this should be paired with Swift temporary URLs.

With the token is available in memory in the agent, the token is embedded with `heartbeat` operations to the ironic API endpoint. This enables the API to authenticate the heartbeat request, and refuse heartbeat requests from the `ironic-python-agent`. As of the Victoria release, use of Agent Token is required for all agents and the previously available setting to force this functionality to be mandatory, `[DEFAULT]require_agent_token` no longer has any effect.

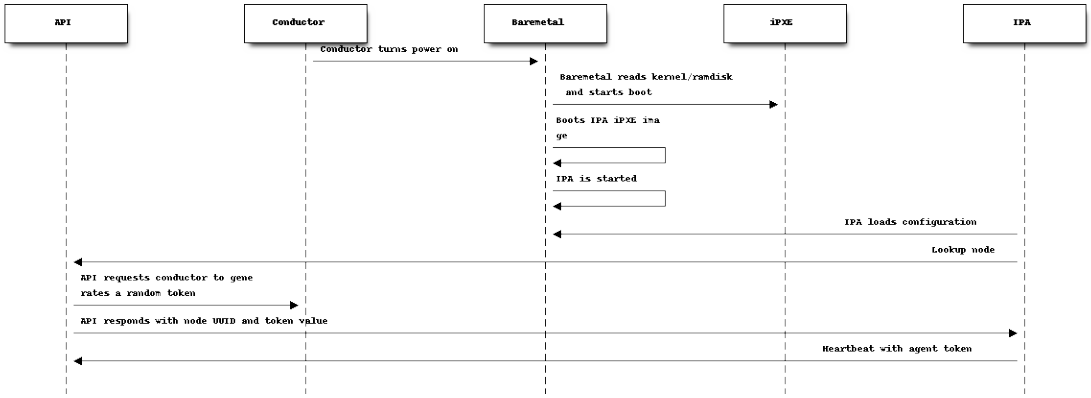
> **Warning:** If the Bare Metal Service is updated, and the version of `ironic-python-agent` should be updated to enable this feature.

In addition to heartbeats being verified, commands from the `ironic-conductor` service to the `ironic-python-agent` also include the token, allowing the agent to authenticate the caller.

With Virtual Media



With PXE/iPXE/etc.



Agent Configuration

An additional setting which may be leveraged with the `ironic-python-agent` is a `agent_token_required` setting. Under normal circumstances, this setting can be asserted via the configuration supplied from the Bare Metal service deployment upon the `lookup` action, but can be asserted via the embedded configuration for the agent in the ramdisk. This setting is also available via kernel command line as `ipa-agent-token-required`.

Deploying without BMC Credentials

The Bare Metal service usually requires BMC credentials for all provisioning operations. Starting with the Victoria release series there is limited support for inspection, cleaning and deployments without the credentials.

Warning: This feature is experimental and only works in a limited scenario. When using it, you have to be prepared to provide BMC credentials in case of a failure or any non-supported actions.

How it works

The expected workflow is as follows:

1. The node is discovered by manually powering it on and gets the *manual-management* hardware type and *agent* power interface.

   If discovery is not used, a node can be enrolled through the API and then powered on manually.

2. The operator moves the node to *manageable*. It works because the *agent* power only requires to be able to connect to the agent.

3. The operator moves the node to *available*. Cleaning happens normally via the already running agent. If reboot is needed, it is done by telling the agent to reboot the node in-band.

4. A user deploys the node. Deployment happens normally via the already running agent.

5. In the end of the deployment, the node is rebooted via the reboot command instead of power off+on.

Enabling

Fast-Track Deployment is a requirement for this feature to work. After enabling it, adds the `agent` power interface and the `manual-management` hardware type to the enabled list:

```
[DEFAULT]
enabled_hardware_types = manual-management
enabled_management_interfaces = noop
enabled_power_interfaces = agent

[deploy]
fast_track = true
```

As usual with the noop management, enable the networking boot fallback:

```
[pxe]
enable_netboot_fallback = true
```

If using discovery, configure discovery in ironic-inspector with the default driver set to `manual-management`.

Limitations

- Only the `noop` network interface is supported.
- Undeploy and rescue are not supported, you need to add BMC credentials first.
- If any errors happens in the process, recovery will likely require BMC credentials.
- Only rebooting is possible through the API, power on/off commands will fail.

Layer 3 or DHCP-less ramdisk booting

Booting nodes via PXE, while universally supported, suffers from one disadvantage: it requires a direct L2 connectivity between the node and the control plane for DHCP. Using virtual media it is possible to avoid not only the unreliable TFTP protocol, but DHCP altogether.

When network data is provided for a node as explained below, the generated virtual media ISO will also serve as a configdrive, and the network data will be stored in the standard OpenStack location.

The simple-init element needs to be used when creating the deployment ramdisk. The Glean tool will look for a media labeled as `config-2`. If found, the network information from it will be read, and the nodes networking stack will be configured accordingly.

```
ironic-python-agent-builder -o /output/ramdisk \
    debian-minimal -e simple-init
```

Warning: Ramdisks based on distributions with NetworkManager require Glean 1.19.0 or newer to work.

Note: If desired, some interfaces can still be configured to use DHCP.

Hardware type support

This feature is known to work with the following hardware types:

- *Redfish* with `redfish-virtual-media` boot
- *iLO* with `ilo-virtual-media` boot

Configuring network data

When the Bare Metal service is running within OpenStack, no additional configuration is required - the network configuration will be fetched from the Network service.

Alternatively, the user can build and pass network configuration in form of a network_data JSON to a node via the `network_data` field. Node-based configuration takes precedence over the configuration generated by the Network service and also works in standalone mode.

```
baremetal node set --network-data ~/network_data.json <node>
```

An example network data:

```
{
    "links": [
        {
            "id": "port-92750f6c-60a9-4897-9cd1-090c5f361e18",
            "type": "phy",
            "ethernet_mac_address": "52:54:00:d3:6a:71"
        }
    ],
    "networks": [
        {
            "id": "network0",
            "type": "ipv4",
            "link": "port-92750f6c-60a9-4897-9cd1-090c5f361e18",
            "ip_address": "192.168.122.42",
            "netmask": "255.255.255.0",
            "network_id": "network0",
            "routes": []
        }
    ],
    "services": []
}
```

Note: Some fields are redundant with the port information. Were looking into simplifying the format, but currently all these fields are mandatory.

Youll need the deployed image to support network data, e.g. by pre-installing cloud-init or Glean on it (most cloud images have the former). Then you can provide the network data when deploying, for example:

```
baremetal node deploy <node> \
    --config-drive "{\"network_data\": $(cat ~/network_data.json)}"
```

Some first-boot services, such as Ignition, dont support network data. You can provide their configuration as part of user data instead:

```
baremetal node deploy <node> \
    --config-drive "{\"user_data\": \"... ignition config ...\"}"
```

Deploying outside of the provisioning network

If you need to combine traditional deployments using a provisioning network with virtual media deployments over L3, you may need to provide an alternative IP address for the remote nodes to connect to:

```
[deploy]
http_url = <HTTP server URL internal to the provisioning network>
external_http_url = <HTTP server URL with a routable IP address>
```

You may also need to override the callback URL, which is normally fetched from the service catalog or configured in the `[service_catalog]` section:

```
[deploy]
external_callback_url = <Bare Metal API URL with a routable IP address>
```

Tuning Ironic

Memory Utilization

Memory utilization is a difficult thing to tune in Ironic as largely we may be asked by API consumers to perform work for which the underlying tools require large amounts of memory.

The biggest example of this is image conversion. Images not in a raw format need to be written out to disk for conversion (when requested) which requires the conversion process to generate an in-memory map to re-assemble the image contents into a coherent stream of data. This entire process also stresses the kernel buffers and cache.

This ultimately comes down to a trade-off of Memory versus Performance, similar to the trade-off of Performance versus Cost.

On a plus side, an idle Ironic deployment does not need much in the way of memory. On the down side, a highly bursty environment where a large number of concurrent deployments may be requested should consider two aspects:

- How is the ironic-api service/process set up? Will more processes be launched automatically?
- Are images prioritized for storage size on disk? Or are they compressed and require format conversion?

API

Ironics API should have a fairly stable memory footprint with activity, however depending on how the webserver is running the API, additional processes can be launched.

Under normal conditions, as of Ironic 15.1, the `ironic-api` service/process consumes approximately 270MB of memory per worker. Depending on how the process is being launched, the number of workers and maximum request threads per worker may differ. Naturally there are configuration and performance trade-offs.

- Directly as a native python process, i.e. execute `ironic-api` processes. Each single worker allows for multiple requests to be handled and threaded at the same time which can allow high levels of

request concurrency. As of the Victoria cycle, a direct invocation of the `ironic-api` program will only launch a maximum of four workers.

- Launched via a wrapper such as Apache+uWSGI may allow for multiple distinct worker processes, but these workers typically limit the number of request processing threads that are permitted to execute. This means requests can stack up in the front-end webserver and be released to the `ironic-api` as prior requests complete. In environments with long running synchronous calls, such as use of the vendor passthru interface, this can be very problematic.

When the webserver is launched by the API process directly, the default is based upon the number of CPU sockets in your machine.

When launching using uwsgi, this will entirely vary upon your configuration, but balancing workers/threads based upon your load and needs is highly advisable. Each worker process is unique and consumes far more memory than a comparable number of worker threads. At the same time, the scheduler will focus on worker processes as the threads are greenthreads.

Note: Host operating systems featuring in-memory de-duplication should see an improvement in the overall memory footprint with multiple processes, but this is not something the development team has measured and will vary based upon multiple factors.

One important item to note: each Ironic API service/process *does* keep a copy of the hash ring as generated from the database *in-memory*. This is done to help allocate load across a cluster in-line with how individual nodes and their responsible conductors are allocated across the cluster. In other words, your amount of memory WILL increase corresponding to the number of nodes managed by each ironic conductor. It is important to understand that features such as conductor groups means that only matching portions of nodes will be considered for the hash ring if needed.

Conductor

A conductor process will launch a number of other processes, as required, in order to complete the requested work. Ultimately this means it can quickly consume large amounts of memory because it was asked to complete a substantial amount of work all at once.

The `ironic-conductor` from ironic 15.1 consumes by default about 340MB of RAM in an idle configuration. This process, by default, operates as a single process. Additional processes can be launched, but they must have unique resolvable hostnames and addresses for JSON-RPC or use a central oslo.messaging supported message bus in order for Webserver API to Conductor API communication to be functional.

Typically, the most memory intensive operation that can be triggered is a image conversion for deployment, which is limited to 1GB of RAM per conversion process.

Most deployments, by default, do have a concurrency limit depending on their Compute (See nova.conf setting `max_concurrent_builds`) configuration. However, this is only per `nova-compute` worker, so naturally this concurrency will scale with additional workers.

Stand-alone users can easily request deployments exceeding the Compute service default maximum concurrent builds. As such, if your environment is used this way, you may wish to carefully consider your deployment architecture.

With a single nova-compute process talking to a single conductor, asked to perform ten concurrent deployments of images requiring conversion, the memory needed may exceed 10GB. This does however, entirely depend upon image block structure and layout, and what deploy interface is being used.

Database

Query load upon the database is one of the biggest potential bottlenecks which can cascade across a deployment and ultimately degrade service to an Ironic user.

Often, depending on load, query patterns, periodic tasks, and so on and so forth, additional indexes may be needed to help provide hints to the database so it can most efficently attempt to reduce the number of rows which need to be examined in order to return a result set.

Adding indexes

This example below is specific to MariaDB/MySQL, but the syntax should be easy to modify for operators using PostgreSQL.

```
use ironic;
create index owner_idx on nodes (owner) LOCK = SHARED;
create index lessee_idx on nodes (lessee) LOCK = SHARED;
create index driver_idx on nodes (driver) LOCK = SHARED;
create index provision_state_idx on nodes (provision_state) LOCK = SHARED;
create index reservation_idx on nodes (reservation) LOCK = SHARED;
create index conductor_group_idx on nodes (conductor_group) LOCK = SHARED;
create index resource_class_idx on nodes (resource_class) LOCK = SHARED;
```

Note: The indexes noted have been added automatically by Xena versions of Ironic and later. They are provided here as an example and operators can add them manually prior with versions of Ironic. The database upgrade for the Xena release of Ironic which adds these indexes are only aware of being able to skip index creation if it already exists on MySQL/MariaDB.

Note: It may be possible to use LOCK = NONE. Basic testing indicates this takes a little bit longer, but shouldnt result in the database table becoming write locked during the index creation. If the database engine cannot support this, then the index creation will fail.

Database platforms also have a concept of what is called a compound index where the index is aligned with the exact query pattern being submitted to the database. The database is able to use this compound index to attempt to drastically reduce the result set generation time for the remainder of the query. As of the composition of this document, we do not ship compound indexes in Ironic as we feel the most general benefit is single column indexes, and depending on data present, an operator may wish to explore compound indexes with their database administrator, as comound indexes can also have negative performance impacts if improperly constructed.

```
use ironic;
create index my_custom_app_query_index on nodes (reservation, provision_state,
↪ driver);
```

The risk, and *WHY* you should engage a Database Administrator, is depending on your configuration, the actual index may need to include one or more additional fields such as owner or lessee which may be added on to the index. At the same time, queries with less field matches, or in different orders will exhibit different performance as the compound index may not be able to be consulted.

Indexes will not fix everything

Indexes are not a magical cure-all for all API or database performance issues, but they are an increadibly important part depending on data access and query patterns.

The underlying object layer and data conversions including record pagination do add a substantial amount of overhead to what may otherwise return as a result set on a manual database query. In Ironics case, due to the object model and the need to extract multiple pieces of data at varying levels of the data model to handle cases such as upgrades, the entire result set is downloaded and transformed which is an overhead you do not experience with a command line database client.

BMC interaction

In its default configuration, Ironic runs a periodic task to synchronize the power state of the managed physical nodes with the Ironic database. For the hardware type `ipmi` (see *IPMI driver*) and depending on the number of nodes, the network connectivity, and the parallelism of these queries, this synchronization may fail and retries will be triggered. Please find more details on the power synchronization and which options to adapt in case too many power sync failures occur in the section on *Power Synchronization*.

What can I do?

Previously in this document, weve already suggested some architectural constraints and limitations, but there are some things that can be done to maximize performance. Again, this will vary greatly depending on your use.

- Use the `direct` deploy interface. This offloads any final image conversion to the host running the `ironic-python-agent`. Additionally, if Swift or other object storage such as RadosGW is used, downloads can be completely separated from the host running the `ironic-conductor`.
- Use small/compact raw images. Qcow2 files are generally compressed and require substantial amounts of memory to decompress and stream.
- Tune the internal memory limit for the conductor using the `[DEFAULT]memory_required_minimum` setting. This will help the conductor throttle back memory intensive operations. The default should prevent Out-of-Memory operations, but under extreme memory pressure this may still be sub-optimal. Before changing this setting, it is highly advised to consult with your resident Unix wizard or even the Ironic development team in upstream IRC. This feature was added in the Wallaby development cycle.
- If network bandwidth is the problem you are seeking to solve for, you may wish to explore a mix of the `direct` deploy interface and caching proxies. Such a configuration can be highly beneficial in wide area deployments. See *Using proxies for image download*.
- If youre making use of large configuration drives, you may wish to ensure youre using Swift to store them as opposed to housing them inside of the database. The entire object and contents are returned whenever Ironic needs to evaluate the entire node, which can become a performance impact. For more information on configuration drives, please see *Enabling the configuration drive*.

Secure RBAC

Suggested Reading

It is likely an understatement to say that policy enforcement is a complex subject. It requires operational context to craft custom policy to meet general use needs. Part of this is why the Secure RBAC effort was started, to provide consistency and a good starting place for most users who need a higher level of granularity.

That being said, it would likely help anyone working to implement customization of these policies to consult some reference material in hopes of understanding the context.

- Keystone Adminstrator Guide - Service API Protection
- Ironic Scoped Role Based Access Control Specification

Historical Context - How we reached our access model

Ironic has reached the access model through an evolution the API and the data stored. Along with the data stored, the enforcement of policy based upon data stored in these fields.

- Ownership Information Storage
- Allow Node owners to Administer
- Allow Leasable Nodes

System Scoped

System scoped authentication is intended for administrative activities such as those crossing tenants/projects, as all tenants/projects should be visible to `system` scoped users in Ironic.

System scoped requests do not have an associated `project_id` value for the Keystone request authorization token utilized to speak with Ironic. These requests are translated through keystonemiddleware into values which tell Ironic what to do. Or to be more precise, tell the policy enforcement framework the information necessary to make decisions.

System scoped requests very much align with the access controls of Ironic before the Secure RBAC effort. The original custom role `baremetal_admin` privileges are identical to a system scoped `admin`s privileges. Similarly `baremetal_observer` is identical to a system scoped `reader`. In these concepts, the `admin` is allowed to create/delete objects/items. The `reader` is allowed to read details about items and is intended for users who may need an account with read-only access for or front-line support purposes.

In addition to these concepts, a `member` role exists in the Secure RBAC use model. Ironic does support this role, and in general `member` role users in a system scope are able to perform basic updates/changes, with the exception of special fields like those to disable cleaning.

Project Scoped

Project scoped authentication is when a request token and associated records indicate an associated `project_id` value.

Legacy Behavior

The legacy behavior of API service is that all requests are treated as project scoped requests where access is governed using an admin project. This behavior is *deprecated*. The new behavior is a delineation of access through `system` scoped and `project` scoped requests.

In essence, what would have served as an admin project, is now `system` scoped usage.

Previously, Ironic API, by default, responded with access denied or permitted based upon the admin project and associated role. These responses would generate an HTTP 403 if the project was incorrect or if a user role.

Note: While Ironic has had the concept of an `owner` and a `lessee`, they are *NOT* used by default. They require custom policy configuration files to be used in the legacy operating mode.

Supported Endpoints

- /nodes
- /nodes/<uuid>/ports
- /nodes/<uuid>/portgroups
- /nodes/<uuid>/volume/connectors
- /nodes/<uuid>/volume/targets
- /nodes/<uuid>/allocation
- /ports
- /portgroups
- /volume/connectors
- /volume/targets
- /allocations

How Project Scoped Works

Ironic has two project use models where access is generally more delegative to an `owner` and access to a `lessee` is generally more utilitarian.

The purpose of an owner, is more to enable the System Operator to delegate much of the administrative activity of a Node to the owner. This may be because they physically own the hardware, or they are in charge of the node. Regardless of the use model that the fields and mechanics support, these fields are to support humans, and possibly services where applicable.

The purpose of a lessee is more for a *tenant* in their *project* to be able to have access to perform basic actions with the API. In some cases that may be to reprovision or rebuild a node. Ultimately that is the lessees prerogative, but by default there are actions and field updates that cannot be performed by default. This is also governed by access level within a project.

These policies are applied in the way data is viewed and how data can be updated. Generally, an inability to view a node is an access permission issue in term of the project ID being correct for owner/lessee.

The ironic project has attempted to generally codify what we believe is reasonable, however operators may wish to override these policy settings. For details general policy setting details, please see *Policies*.

Field value visibility restrictions

Ironics API, by default has a concept of filtering node values to prevent sensitive data from being leaked. System scoped users are subjected to basic restrictions, whereas project scoped users are, by default, examined further and against additional policies. This threshold is controlled with the `baremetal:node:get:filter_threshold`.

By default, the following fields are masked on Nodes and are controlled by the associated policies. By default, owners are able to see insight into the infrastructure, whereas lessee users *CANNOT* view these fields by default.

- `last_error` - `baremetal:node:get:last_error`
- `reservation` - `baremetal:node:get:reservation`
- `driver_internal_info` - `baremetal:node:get:driver_internal_info`
- `driver_info` - `baremetal:node:get:driver_info`

Field update restrictions

Some of the fields in this list are restricted to System scoped users, or even only System Administrators. Some of these default restrictions are likely obvious. Owners cant change the owner. Lessees cant change the owner.

- `driver_info` - `baremetal:node:update:driver_info`
- `properties` - `baremetal:node:update:properties`
- `chassis_uuid` - `baremetal:node:update:chassis_uuid`
- `instance_uuid` - `baremetal:node:update:instance_uuid`
- `lessee` - `baremetal:node:update:lessee`
- `owner` - `baremetal:node:update:owner`

- driver - baremetal:node:update:driver_interfaces
- *_interface - baremetal:node:update:driver_interfaces
- network_data - baremetal:node:update:network_data
- conductor_group - baremetal:node:update:conductor_group
- name - baremetal:node:update:name
- retired - baremetal:node:update:driver_info
- retired_reason - baremetal:node:update:retired

Warning: The chassis_uuid field is a write-once-only field. As such it is restricted to system scoped administrators.

More information is available on these fields in *Policies*.

Allocations

The allocations endpoint of the API is somewhat different than other endpoints as it allows for the allocation of physical machines to an admin. In this context, there is not already an owner or project_id to leverage to control access for the creation process, any project member does have the inherent privilege of requesting an allocation. That being said, their allocation request will require physical nodes to be owned or leased to the project_id through the node fields owner or lessee.

Ability to override the owner is restricted to system scoped users by default and any new allocation being requested with a specific owner, if made in project scope, will have the project_id recorded as the owner of the allocation.

Ultimately, an operational behavior difference exists between the owner and lessee rights in terms of allocations. With the standard access rights, lessee users are able to create allocations if they own nodes which are not allocated or deployed, but they cannot reprovision nodes when using only a member role. This limitation is not the case for project-scoped users with the admin role.

Warning: The allocation endpoints use is restricted to project scoped interactions until [oslo_policy]enforce_new_defaults has been set to True using the baremetal:allocation:create_pre_rbac policy rule. This is in order to prevent endpoint misuse. Afterwards all project scoped allocations will automatically populate an owner. System scoped request are not subjected to this restriction, and operators may change the default restriction via the baremetal:allocation:create_restricted policy.

Practical differences

Most users, upon implementing the use of `system` scoped authentication should not notice a difference as long as their authentication token is properly scoped to `system` and with the appropriate role for their access level. For most users who used a `baremetal` project, or other custom project via a custom policy file, along with a custom role name such as `baremetal_admin`, this will require changing the user to be a `system` scoped user with `admin` privileges.

The most noticeable difference for API consumers is the HTTP 403 access code is now mainly a HTTP 404 access code. The access concept has changed from Does the user broadly have access to the API? to Does user have access to the node, and then do they have access to the specific resource?.

What is an owner or lessee?

An `owner` or `lessee` is the project which has been assigned baremetal resources. Generally these should be service projects as opposed to a project dedicated to a specific user. This will help prevent the need to involve a `system` scoped administrator from having to correct ownership records should a project need to be removed due to an individuals departure.

The underlying `project_id` is used to represent and associate the owner or lessee.

How do I assign an owner?

```
# baremetal node set --owner <project_id> <node>
```

Note: With the default access policy, an `owner` is able to change the assigned `lessee` of a node. However the `lessee` is unable to do the same.

How do I assign a lessee?

```
# baremetal node set --lessee <project_id> <node>
```

What is the difference between an owner and lessee?

This is largely covered in *How Project Scoped Works* although as noted it is largely in means of access. A `lessee` is far more restrictive and an `owner` may revoke access to `lessee`.

Access to the underlying baremetal node is not exclusive between the `owner` and `lessee`, and this use model expects that some level of communication takes place between the appropriate parties.

Deploying with anaconda deploy interface

Ironic supports deploying an OS with the anaconda installer. This anaconda deploy interface works with `pxe` and `ipxe` boot interfaces.

Configuration

The anaconda deploy interface is not enabled by default. To enable this, add `anaconda` to the value of the `enabled_deploy_interfaces` configuration option in ironic.conf. For example:

```
[DEFAULT]
...
enabled_deploy_interfaces = direct,anaconda
...
```

This change takes effect after all the ironic conductors have been restarted.

The default kickstart template is specified via the configuration option `[anaconda]default_ks_template`. It is set to this ks.cfg.template but can be modified to be some other template.

```
[anaconda]
default_ks_template = file:///etc/ironic/ks.cfg.template
```

When creating an ironic node, specify `anaconda` as the deploy interface. For example:

```
baremetal node create --driver ipmi \
    --deploy-interface anaconda \
    --boot-interface ipxe
```

You can also set the anaconda deploy interface via `--deploy-interface` on an existing node:

```
baremetal node set <node> --deploy-interface anaconda
```

Creating an OS Image

While anaconda allows installing individual RPMs, the default kickstart file expects an OS tarball to be used as the OS image.

This `baremetal.yum` file contains all the yum/dnf commands that need to be run in order to generate the OS tarball. These commands install packages and package groups that need to be in the image:

```
group install 'Minimal Install'
install cloud-init
ts run
```

An OS tarball can be created using following set of commands, along with the above `baremetal.yum` file:

```
export CHROOT=/home/<user>/os-image
mkdir -p $(CHROOT)
mkdir -p $(CHROOT)/{dev,proc,run,sys}
chown -hR root:root $(CHROOT)
mount --bind /var/cache/yum $(CHROOT)/var/cache/yum
mount --bind /dev $(CHROOT)/dev
mount -t proc proc $(CHROOT)/proc
mount -t tmpfs tmpfs $(CHROOT)/run
mount -t sysfs sysfs $(CHROOT)/sys
dnf -y --installroot=$(CHROOT) makecache
dnf -y --installroot=$(CHROOT) shell baremetal.yum
rpm --root $(CHROOT) --import $(CHROOT)/etc/pki/rpm-gpg/RPM-GPG-KEY-*
truncate -s 0 $(CHROOT)/etc/machine-id
umount $(CHROOT)/var/cache/yum
umount $(CHROOT)/dev
umount $(CHROOT)/proc
umount $(CHROOT)/run
umount $(CHROOT)/sys
tar cpzf os-image.tar.gz --xattrs --acls --selinux -C $(CHROOT) .
```

Configuring the OS Image in glance

Anaconda is a two-stage installer stage 1 consists of the kernel and ramdisk and stage 2 lives in a squashfs file. All these components can be found in the CentOS/RHEL/Fedora ISO images.

The kernel and ramdisk can be found at `/images/pxeboot/vmlinuz` and `/images/pxeboot/initrd.img` respectively in the ISO. The stage 2 squashfs image can be normally found at `/LiveOS/squashfs.img` or `/images/install.img`.

The OS tarball must be configured with the following properties in glance, in order to be used with the anaconda deploy driver:

- `kernel_id`
- `ramdisk_id`
- `stage2_id`
- `disk_file_extension` (optional)

Valid `disk_file_extension` values are `.img`, `.tar`, `.tbz`, `.tgz`, `.txz`, `.tar.gz`, `.tar.bz2`, and `.tar.xz`. When `disk_file_extension` property is not set to one of the above valid values the anaconda installer will assume that the image provided is a mountable OS disk.

This is an example of adding the anaconda-related images and the OS tarball to glance:

```
openstack image create --file ./vmlinuz --container-format aki \
    --disk-format aki --shared anaconda-kernel-<version>
openstack image create --file ./initrd.img --container-format ari \
    --disk-format ari --shared anaconda-ramdisk-<version>
openstack image create --file ./squashfs.img --container-format ari \
    --disk-format ari --shared anaconda-stage-<verison>
openstack image create --file ./os-image.tar.gz --container-format \
```

(continues on next page)

(continued from previous page)

```
    compressed --disk-format raw --shared \
    --property kernel_id=<glance_uuid_vmlinuz> \
    --property ramdisk_id=<glance_uuid_ramdisk> \
    --property stage2_id=<glance_uuid_stage2> disto-name-version \
    --property disk_file_extension=.tgz
```

Creating a bare metal server

Apart from uploading a custom kickstart template to glance and associating it with the OS image via the `ks_template` property in glance, operators can also set the kickstart template in the ironic nodes `instance_info` field. The kickstart template set in `instance_info` takes precedence over the one specified via the OS image in glance. If no kickstart template is specified (via the nodes `instance_info` or `ks_template` glance image property), the default kickstart template will be used to deploy the OS.

This is an example of how to set the kickstart template for a specific ironic node:

```
openstack baremetal node set <node> \
    --instance_info ks_template=glance://uuid
```

Limitations

This deploy interface has only been tested with Red Hat based operating systems that use anaconda. Other systems are not supported.

Deploy Steps

The deploy steps section has moved to *Node Deployment*.

4.2.26 Dashboard Integration

A plugin for the OpenStack Dashboard (horizon) service is under development. Documentation for that can be found within the ironic-ui project.

- Dashboard (horizon) plugin

CHAPTER

FIVE

CONFIGURATION GUIDE

5.1 Configuration Reference

Many aspects of the Bare Metal service are specific to the environment it is deployed in. The following pages describe configuration options that can be used to adjust the service to your particular situation.

5.1.1 Configuration Options

The following is an overview of all available configuration options in Ironic. For a sample configuration file, refer to *Sample Configuration File*.

DEFAULT

debug

> **Type** boolean
>
> **Default** `False`
>
> **Mutable** This option can be changed without restarting.

If set to true, the logging level will be set to DEBUG instead of the default INFO level.

log_config_append

> **Type** string
>
> **Default** `<None>`
>
> **Mutable** This option can be changed without restarting.

The name of a logging configuration file. This file is appended to any existing logging configuration files. For details about logging configuration files, see the Python logging module documentation. Note that when logging configuration files are used then all logging configuration is set in the configuration file and other logging configuration options are ignored (for example, log-date-format).

Table 1: Deprecated Variations

Group	Name
DEFAULT	log-config
DEFAULT	log_config

`log_date_format`

Type string

Default `%Y-%m-%d %H:%M:%S`

Defines the format string for %(asctime)s in log records. Default: the value above . This option is ignored if log_config_append is set.

`log_file`

Type string

Default `<None>`

(Optional) Name of log file to send logging output to. If no default is set, logging will go to stderr as defined by use_stderr. This option is ignored if log_config_append is set.

Table 2: Deprecated Variations

Group	Name
DEFAULT	logfile

`log_dir`

Type string

Default `<None>`

(Optional) The base directory used for relative log_file paths. This option is ignored if log_config_append is set.

Table 3: Deprecated Variations

Group	Name
DEFAULT	logdir

`watch_log_file`

Type boolean

Default `False`

Uses logging handler designed to watch file system. When log file is moved or removed this handler will open a new log file with specified path instantaneously. It makes sense only if log_file option is specified and Linux platform is used. This option is ignored if log_config_append is set.

`use_syslog`

Type boolean

Default `False`

Use syslog for logging. Existing syslog format is DEPRECATED and will be changed later to honor RFC5424. This option is ignored if log_config_append is set.

`use_journal`

Type boolean

Default `False`

Enable journald for logging. If running in a systemd environment you may wish to enable journal support. Doing so will use the journal native protocol which includes structured metadata in addition to log messages.This option is ignored if log_config_append is set.

`syslog_log_facility`

Type string

Default `LOG_USER`

Syslog facility to receive log lines. This option is ignored if log_config_append is set.

`use_json`

Type boolean

Default `False`

Use JSON formatting for logging. This option is ignored if log_config_append is set.

`use_stderr`

Type boolean

Default `False`

Log output to standard error. This option is ignored if log_config_append is set.

`use_eventlog`

Type boolean

Default `False`

Log output to Windows Event Log.

`log_rotate_interval`

Type integer

Default `1`

The amount of time before the log files are rotated. This option is ignored unless log_rotation_type is set to interval.

`log_rotate_interval_type`

Type string

Default `days`

Valid Values Seconds, Minutes, Hours, Days, Weekday, Midnight

Rotation interval type. The time of the last file change (or the time when the service was started) is used when scheduling the next rotation.

`max_logfile_count`

Type integer

Default `30`

Maximum number of rotated log files.

`max_logfile_size_mb`

Type integer

Default `200`

Log file maximum size in MB. This option is ignored if log_rotation_type is not set to size.

`log_rotation_type`

Type string

Default none

Valid Values interval, size, none

Log rotation type.

Possible values

interval Rotate logs at predefined time intervals.

size Rotate logs once they reach a predefined size.

none Do not rotate log files.

`logging_context_format_string`

Type string

Default `%(asctime)s.%(msecs)03d %(process)d %(levelname)s %(name)s [%(request_id)s %(user_identity)s] %(instance)s%(message)s`

Format string to use for log messages with context. Used by oslo_log.formatters.ContextFormatter

`logging_default_format_string`

Type string

Default `%(asctime)s.%(msecs)03d %(process)d %(levelname)s %(name)s [-] %(instance)s%(message)s`

Format string to use for log messages when context is undefined. Used by oslo_log.formatters.ContextFormatter

`logging_debug_format_suffix`

Type string

Default `%(funcName)s %(pathname)s:%(lineno)d`

Additional data to append to log message when logging level for the message is DEBUG. Used by oslo_log.formatters.ContextFormatter

`logging_exception_prefix`

Type string

Default `%(asctime)s.%(msecs)03d %(process)d ERROR %(name)s %(instance)s`

Prefix each line of exception output with this format. Used by oslo_log.formatters.ContextFormatter

`logging_user_identity_format`

Type string

Default `%(user)s %(project)s %(domain)s %(user_domain)s %(project_domain)s`

Defines the format string for %(user_identity)s that is used in logging_context_format_string. Used by oslo_log.formatters.ContextFormatter

`default_log_levels`

Type list

Default `['amqp=WARNING', 'amqplib=WARNING', 'qpid.messaging=INFO', 'oslo.messaging=INFO', 'oslo_messaging=INFO', 'sqlalchemy=WARNING', 'stevedore=INFO', 'eventlet.wsgi.server=INFO', 'iso8601=WARNING', 'requests=WARNING', 'glanceclient=WARNING', 'urllib3.connectionpool=WARNING', 'keystonemiddleware.auth_token=INFO', 'keystoneauth.session=INFO', 'openstack=WARNING', 'oslo_policy=WARNING']`

List of package logging levels in logger=LEVEL pairs. This option is ignored if log_config_append is set.

`publish_errors`

Type boolean

Default `False`

Enables or disables publication of error events.

`instance_format`

Type string

Default `"[instance: %(uuid)s] "`

The format for an instance that is passed with the log message.

instance_uuid_format

Type string

Default `"[instance: %(uuid)s] "`

The format for an instance UUID that is passed with the log message.

rate_limit_interval

Type integer

Default `0`

Interval, number of seconds, of log rate limiting.

rate_limit_burst

Type integer

Default `0`

Maximum number of logged messages per rate_limit_interval.

rate_limit_except_level

Type string

Default `CRITICAL`

Log level name used by rate limiting: CRITICAL, ERROR, INFO, WARNING, DEBUG or empty string. Logs with level greater or equal to rate_limit_except_level are not filtered. An empty string means that all levels are filtered.

fatal_deprecations

Type boolean

Default `False`

Enables or disables fatal status of deprecations.

rpc_conn_pool_size

Type integer

Default `30`

Minimum Value 1

Size of RPC connection pool.

Table 4: Deprecated Variations

Group	Name
DEFAULT	rpc_conn_pool_size

conn_pool_min_size

Type integer

Default `2`

The pool size limit for connections expiration policy

`conn_pool_ttl`

Type integer

Default `1200`

The time-to-live in sec of idle connections in the pool

`executor_thread_pool_size`

Type integer

Default `64`

Size of executor thread pool when executor is threading or eventlet.

Table 5: Deprecated Variations

Group	Name
DEFAULT	rpc_thread_pool_size

`rpc_response_timeout`

Type integer

Default `60`

Seconds to wait for a response from a call.

`transport_url`

Type string

Default `rabbit://`

The network address and optional user credentials for connecting to the messaging backend, in URL format. The expected format is:

driver://[user:pass@]host:port[,[userN:passN@]hostN:portN]/virtual_host?query

Example: rabbit://rabbitmq:password@127.0.0.1:5672//

For full details on the fields in the URL see the documentation of oslo_messaging.TransportURL at https://docs.openstack.org/oslo.messaging/latest/reference/transport.html

`control_exchange`

Type string

Default `openstack`

The default exchange under which topics are scoped. May be overridden by an exchange name specified in the transport_url option.

rpc_ping_enabled

Type boolean

Default `False`

Add an endpoint to answer to ping calls. Endpoint is named oslo_rpc_server_ping

run_external_periodic_tasks

Type boolean

Default `True`

Some periodic tasks can be run in a separate process. Should we run them here?

backdoor_port

Type string

Default `<None>`

Enable eventlet backdoor. Acceptable values are 0, <port>, and <start>:<end>, where 0 results in listening on a random tcp port number; <port> results in listening on the specified port number (and not enabling backdoor if that port is in use); and <start>:<end> results in listening on the smallest unused port number within the specified range of port numbers. The chosen port is displayed in the services log file.

backdoor_socket

Type string

Default `<None>`

Enable eventlet backdoor, using the provided path as a unix socket that can receive connections. This option is mutually exclusive with backdoor_port in that only one should be provided. If both are provided then the existence of this option overrides the usage of that option. Inside the path {pid} will be replaced with the PID of the current process.

log_options

Type boolean

Default `True`

Enables or disables logging values of all registered options when starting a service (at DEBUG level).

graceful_shutdown_timeout

Type integer

Default `60`

Specify a timeout after which a gracefully shutdown server will exit. Zero value means endless wait.

`auth_strategy`

Type string

Default `keystone`

Valid Values noauth, keystone, http_basic

Authentication strategy used by ironic-api. noauth should not be used in a production environment because all authentication will be disabled.

Possible values

noauth no authentication

keystone use the Identity service for authentication

http_basic HTTP basic authentication

`http_basic_auth_user_file`

Type string

Default `/etc/ironic/htpasswd`

Path to Apache format user authentication file used when auth_strategy=http_basic

`debug_tracebacks_in_api`

Type boolean

Default `False`

Return server tracebacks in the API response for any error responses. WARNING: this is insecure and should not be used in a production environment.

`pecan_debug`

Type boolean

Default `False`

Enable pecan debug mode. WARNING: this is insecure and should not be used in a production environment.

`default_resource_class`

Type string

Default `<None>`

Mutable This option can be changed without restarting.

Resource class to use for new nodes when no resource class is provided in the creation request.

enabled_hardware_types

Type list

Default `['ipmi', 'redfish']`

Specify the list of hardware types to load during service initialization. Missing hardware types, or hardware types which fail to initialize, will prevent the conductor service from starting. This option defaults to a recommended set of production-oriented hardware types. A complete list of hardware types present on your system may be found by enumerating the ironic.hardware.types entrypoint.

enabled_bios_interfaces

Type list

Default `['no-bios', 'redfish']`

Specify the list of bios interfaces to load during service initialization. Missing bios interfaces, or bios interfaces which fail to initialize, will prevent the ironic-conductor service from starting. At least one bios interface that is supported by each enabled hardware type must be enabled here, or the ironic-conductor service will not start. Must not be an empty list. The default value is a recommended set of production-oriented bios interfaces. A complete list of bios interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.bios entrypoint. When setting this value, please make sure that every enabled hardware type will have the same set of enabled bios interfaces on every ironic-conductor service.

default_bios_interface

Type string

Default `<None>`

Default bios interface to be used for nodes that do not have bios_interface field set. A complete list of bios interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.bios entrypoint.

enabled_boot_interfaces

Type list

Default `['ipxe', 'pxe', 'redfish-virtual-media']`

Specify the list of boot interfaces to load during service initialization. Missing boot interfaces, or boot interfaces which fail to initialize, will prevent the ironic-conductor service from starting. At least one boot interface that is supported by each enabled hardware type must be enabled here, or the ironic-conductor service will not start. Must not be an empty list. The default value is a recommended set of production-oriented boot interfaces. A complete list of boot interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.boot entrypoint. When setting this value, please make sure that every enabled hardware type will have the same set of enabled boot interfaces on every ironic-conductor service.

default_boot_interface

Type string

Default <None>

Default boot interface to be used for nodes that do not have boot_interface field set. A complete list of boot interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.boot entrypoint.

`enabled_console_interfaces`

Type list

Default ['no-console']

Specify the list of console interfaces to load during service initialization. Missing console interfaces, or console interfaces which fail to initialize, will prevent the ironic-conductor service from starting. At least one console interface that is supported by each enabled hardware type must be enabled here, or the ironic-conductor service will not start. Must not be an empty list. The default value is a recommended set of production-oriented console interfaces. A complete list of console interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.console entrypoint. When setting this value, please make sure that every enabled hardware type will have the same set of enabled console interfaces on every ironic-conductor service.

`default_console_interface`

Type string

Default <None>

Default console interface to be used for nodes that do not have console_interface field set. A complete list of console interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.console entrypoint.

`enabled_deploy_interfaces`

Type list

Default ['direct']

Specify the list of deploy interfaces to load during service initialization. Missing deploy interfaces, or deploy interfaces which fail to initialize, will prevent the ironic-conductor service from starting. At least one deploy interface that is supported by each enabled hardware type must be enabled here, or the ironic-conductor service will not start. Must not be an empty list. The default value is a recommended set of production-oriented deploy interfaces. A complete list of deploy interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.deploy entrypoint. When setting this value, please make sure that every enabled hardware type will have the same set of enabled deploy interfaces on every ironic-conductor service.

`default_deploy_interface`

Type string

Default <None>

Default deploy interface to be used for nodes that do not have deploy_interface field set. A complete list of deploy interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.deploy entrypoint.

enabled_inspect_interfaces

Type list

Default `['no-inspect', 'redfish']`

Specify the list of inspect interfaces to load during service initialization. Missing inspect interfaces, or inspect interfaces which fail to initialize, will prevent the ironic-conductor service from starting. At least one inspect interface that is supported by each enabled hardware type must be enabled here, or the ironic-conductor service will not start. Must not be an empty list. The default value is a recommended set of production-oriented inspect interfaces. A complete list of inspect interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.inspect entrypoint. When setting this value, please make sure that every enabled hardware type will have the same set of enabled inspect interfaces on every ironic-conductor service.

default_inspect_interface

Type string

Default <None>

Default inspect interface to be used for nodes that do not have inspect_interface field set. A complete list of inspect interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.inspect entrypoint.

enabled_management_interfaces

Type list

Default <None>

Specify the list of management interfaces to load during service initialization. Missing management interfaces, or management interfaces which fail to initialize, will prevent the ironic-conductor service from starting. At least one management interface that is supported by each enabled hardware type must be enabled here, or the ironic-conductor service will not start. Must not be an empty list. The default value is a recommended set of production-oriented management interfaces. A complete list of management interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.management entrypoint. When setting this value, please make sure that every enabled hardware type will have the same set of enabled management interfaces on every ironic-conductor service.

default_management_interface

Type string

Default <None>

Default management interface to be used for nodes that do not have management_interface field set. A complete list of management interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.management entrypoint.

enabled_network_interfaces

Type list

Default `['flat', 'noop']`

Specify the list of network interfaces to load during service initialization. Missing network interfaces, or network interfaces which fail to initialize, will prevent the ironic-conductor service from starting. At least one network interface that is supported by each enabled hardware type must be enabled here, or the ironic-conductor service will not start. Must not be an empty list. The default value is a recommended set of production-oriented network interfaces. A complete list of network interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.network entrypoint. When setting this value, please make sure that every enabled hardware type will have the same set of enabled network interfaces on every ironic-conductor service.

`default_network_interface`

Type string

Default `<None>`

Default network interface to be used for nodes that do not have network_interface field set. A complete list of network interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.network entrypoint.

`enabled_power_interfaces`

Type list

Default `<None>`

Specify the list of power interfaces to load during service initialization. Missing power interfaces, or power interfaces which fail to initialize, will prevent the ironic-conductor service from starting. At least one power interface that is supported by each enabled hardware type must be enabled here, or the ironic-conductor service will not start. Must not be an empty list. The default value is a recommended set of production-oriented power interfaces. A complete list of power interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.power entrypoint. When setting this value, please make sure that every enabled hardware type will have the same set of enabled power interfaces on every ironic-conductor service.

`default_power_interface`

Type string

Default `<None>`

Default power interface to be used for nodes that do not have power_interface field set. A complete list of power interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.power entrypoint.

`enabled_raid_interfaces`

Type list

Default `['agent', 'no-raid', 'redfish']`

Specify the list of raid interfaces to load during service initialization. Missing raid interfaces, or raid interfaces which fail to initialize, will prevent the ironic-conductor service from starting. At least one raid interface that is supported by each enabled hardware type must be enabled here, or the ironic-conductor service will not start. Must not be an empty list. The default value is a recommended set of production-oriented raid interfaces. A complete list of raid interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.raid entrypoint. When setting this value, please make sure that every enabled hardware type will have the same set of enabled raid interfaces on every ironic-conductor service.

default_raid_interface

Type string

Default <None>

Default raid interface to be used for nodes that do not have raid_interface field set. A complete list of raid interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.raid entrypoint.

enabled_rescue_interfaces

Type list

Default ['no-rescue']

Specify the list of rescue interfaces to load during service initialization. Missing rescue interfaces, or rescue interfaces which fail to initialize, will prevent the ironic-conductor service from starting. At least one rescue interface that is supported by each enabled hardware type must be enabled here, or the ironic-conductor service will not start. Must not be an empty list. The default value is a recommended set of production-oriented rescue interfaces. A complete list of rescue interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.rescue entrypoint. When setting this value, please make sure that every enabled hardware type will have the same set of enabled rescue interfaces on every ironic-conductor service.

default_rescue_interface

Type string

Default <None>

Default rescue interface to be used for nodes that do not have rescue_interface field set. A complete list of rescue interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.rescue entrypoint.

enabled_storage_interfaces

Type list

Default ['cinder', 'noop']

Specify the list of storage interfaces to load during service initialization. Missing storage interfaces, or storage interfaces which fail to initialize, will prevent the ironic-conductor service from starting. At least one storage interface that is supported by each enabled hardware type must be enabled here, or the ironic-conductor service will not start. Must not be an empty list. The default value

is a recommended set of production-oriented storage interfaces. A complete list of storage interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.storage entrypoint. When setting this value, please make sure that every enabled hardware type will have the same set of enabled storage interfaces on every ironic-conductor service.

default_storage_interface

Type string

Default `noop`

Default storage interface to be used for nodes that do not have storage_interface field set. A complete list of storage interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.storage entrypoint.

enabled_vendor_interfaces

Type list

Default `['ipmitool', 'redfish', 'no-vendor']`

Specify the list of vendor interfaces to load during service initialization. Missing vendor interfaces, or vendor interfaces which fail to initialize, will prevent the ironic-conductor service from starting. At least one vendor interface that is supported by each enabled hardware type must be enabled here, or the ironic-conductor service will not start. Must not be an empty list. The default value is a recommended set of production-oriented vendor interfaces. A complete list of vendor interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.vendor entrypoint. When setting this value, please make sure that every enabled hardware type will have the same set of enabled vendor interfaces on every ironic-conductor service.

default_vendor_interface

Type string

Default `<None>`

Default vendor interface to be used for nodes that do not have vendor_interface field set. A complete list of vendor interfaces present on your system may be found by enumerating the ironic.hardware.interfaces.vendor entrypoint.

log_in_db_max_size

Type integer

Default `4096`

Max number of characters of any node last_error/maintenance_reason pushed to database.

hash_partition_exponent

Type integer

Default `5`

Exponent to determine number of hash partitions to use when distributing load across conductors. Larger values will result in more even distribution of load and less load when rebalancing the ring,

but more memory usage. Number of partitions per conductor is (2^hash_partition_exponent). This determines the granularity of rebalancing: given 10 hosts, and an exponent of the 2, there are 40 partitions in the ring.A few thousand partitions should make rebalancing smooth in most cases. The default is suitable for up to a few hundred conductors. Configuring for too many partitions has a negative impact on CPU usage.

hash_ring_reset_interval

Type integer

Default `15`

Time (in seconds) after which the hash ring is considered outdated and is refreshed on the next access.

hash_ring_algorithm

Type string

Default `md5`

Valid Values shake_256, sha256, sha3_512, shake_128, sha1, sha512, blake2s, sha3_224, sha3_256, sha3_384, sha224, blake2b, md5, sha384

Advanced Option Intended for advanced users and not used by the majority of users, and might have a significant effect on stability and/or performance.

Hash function to use when building the hash ring. If running on a FIPS system, do not use md5. WARNING: all ironic services in a cluster MUST use the same algorithm at all times. Changing the algorithm requires an offline update.

force_raw_images

Type boolean

Default `True`

Mutable This option can be changed without restarting.

If True, convert backing images to raw disk image format.

raw_image_growth_factor

Type floating point

Default `2.0`

Minimum Value 1.0

The scale factor used for estimating the size of a raw image converted from compact image formats such as QCOW2. Default is 2.0, must be greater than 1.0.

isolinux_bin

Type string

Default `/usr/lib/syslinux/isolinux.bin`

Path to isolinux binary file.

isolinux_config_template

Type string

Default `$pybasedir/common/isolinux_config.template`

Template file for isolinux configuration file.

grub_config_path

Type string

Default `/boot/grub/grub.cfg`

GRUB2 configuration file location on the UEFI ISO images produced by ironic. The default value is usually incorrect and should not be relied on. If you use a GRUB2 image from a certain distribution, use a distribution-specific path here, e.g. EFI/ubuntu/grub.cfg

grub_config_template

Type string

Default `$pybasedir/common/grub_conf.template`

Template file for grub configuration file.

ldlinux_c32

Type string

Default `<None>`

Path to ldlinux.c32 file. This file is required for syslinux 5.0 or later. If not specified, the file is looked for in /usr/lib/syslinux/modules/bios/ldlinux.c32 and /usr/share/syslinux/ldlinux.c32.

esp_image

Type string

Default `<None>`

Path to EFI System Partition image file. This file is recommended for creating UEFI bootable ISO images efficiently. ESP image should contain a FAT12/16/32-formatted file system holding EFI boot loaders (e.g. GRUB2) for each hardware architecture ironic needs to boot. This option is only used when neither ESP nor ISO deploy image is configured to the node being deployed in which case ironic will attempt to fetch ESP image from the configured location or extract ESP image from UEFI-bootable deploy ISO image.

parallel_image_downloads

Type boolean

Default `True`

Mutable This option can be changed without restarting.

Run image downloads and raw format conversions in parallel.

> **Warning:** This option is deprecated for removal. Its value may be silently ignored in the future.
>
> **Reason** Use image_download_concurrency

`image_download_concurrency`

Type integer

Default `20`

Minimum Value 1

How many image downloads and raw format conversions to run in parallel. Only affects image caches.

`my_ip`

Type string

Default `127.0.0.1`

This option has a sample default set, which means that its actual default value may vary from the one documented above.

IPv4 address of this host. If unset, will determine the IP programmatically. If unable to do so, will use 127.0.0.1. NOTE: This field does accept an IPv6 address as an override for templates and URLs, however it is recommended that [DEFAULT]my_ipv6 is used along with DNS names for service URLs for dual-stack environments.

`my_ipv6`

Type string

Default `2001:db8::1`

This option has a sample default set, which means that its actual default value may vary from the one documented above.

IP address of this host using IPv6. This value must be supplied via the configuration and cannot be adequately programmatically determined like the [DEFAULT]my_ip parameter for IPv4.

`notification_level`

Type string

Default `<None>`

Valid Values debug, info, warning, error, critical

Specifies the minimum level for which to send notifications. If not set, no notifications will be sent. The default is for this option to be unset.

Possible values

debug debug level

info info level

warning warning level

error error level

critical critical level

`versioned_notifications_topics`

Type list

Default `['ironic_versioned_notifications']`

Specifies the topics for the versioned notifications issued by Ironic.

The default value is fine for most deployments and rarely needs to be changed. However, if you have a third-party service that consumes versioned notifications, it might be worth getting a topic for that service. Ironic will send a message containing a versioned notification payload to each topic queue in this list.

The list of versioned notifications is visible in https://docs.openstack.org/ironic/latest/admin/notifications.html

`pybasedir`

Type string

Default `/usr/lib/python/site-packages/ironic/ironic`

This option has a sample default set, which means that its actual default value may vary from the one documented above.

Directory where the ironic python module is installed.

`bindir`

Type string

Default `$pybasedir/bin`

Directory where ironic binaries are installed.

`state_path`

Type string

Default `$pybasedir`

Top-level directory for maintaining ironics state.

`default_portgroup_mode`

Type string

Default `active-backup`

Mutable This option can be changed without restarting.

Default mode for portgroups. Allowed values can be found in the linux kernel documentation on bonding: https://www.kernel.org/doc/Documentation/networking/bonding.txt.

`host`

Type string

Default `localhost`

This option has a sample default set, which means that its actual default value may vary from the one documented above.

Name of this node. This can be an opaque identifier. It is not necessarily a hostname, FQDN, or IP address. However, the node name must be valid within an AMQP key, and if using ZeroMQ (will be removed in the Stein release), a valid hostname, FQDN, or IP address.

`pin_release_version`

Type string

Default `<None>`

Valid Values yoga, xena, 9.2, 20.1, 20.0, 19.0, 18.2, 18.1, 18.0, 17.0, 16.2, 16.1, 16.0, 15.1, 15.0, 14.0, 13.0, 12.2, 12.1, 12.0, 11.1, 11.0, 10.1, 10.0

Mutable This option can be changed without restarting.

Used for rolling upgrades. Setting this option downgrades (or pins) the Bare Metal API, the internal ironic RPC communication, and the database objects to their respective versions, so they are compatible with older services. When doing a rolling upgrade from version N to version N+1, set (to pin) this to N. To unpin (default), leave it unset and the latest versions will be used.

Possible values

yoga yoga release

xena xena release

9.2 9.2 release

20.1 20.1 release

20.0 20.0 release

19.0 19.0 release

18.2 18.2 release

18.1 18.1 release

18.0 18.0 release

17.0 17.0 release

16.2 16.2 release

16.1 16.1 release

16.0 16.0 release

15.1 15.1 release

15.0 15.0 release

14.0 14.0 release

13.0 13.0 release

12.2 12.2 release

12.1 12.1 release

12.0 12.0 release

11.1 11.1 release

11.0 11.0 release

10.1 10.1 release

10.0 10.0 release

`rpc_transport`

Type string

Default `oslo`

Valid Values oslo, json-rpc, none

Which RPC transport implementation to use between conductor and API services

Possible values

oslo use oslo.messaging transport

json-rpc use JSON RPC transport

none No RPC, only use local conductor

`minimum_memory_warning_only`

Type boolean

Default `False`

Mutable This option can be changed without restarting.

Setting to govern if Ironic should only warn instead of attempting to hold back the request in order to prevent the exhaustion of system memory.

`minimum_required_memory`

Type integer

Default `1024`

Mutable This option can be changed without restarting.

Minimum memory in MiB for the system to have available prior to starting a memory intensive process on the conductor.

`minimum_memory_wait_time`

Type integer

Default `15`

Mutable This option can be changed without restarting.

Seconds to wait between retries for free memory before launching the process. This, combined with `memory_wait_retries` allows the conductor to determine how long we should attempt to directly retry.

`minimum_memory_wait_retries`

Type integer

Default `6`

Mutable This option can be changed without restarting.

Number of retries to hold onto the worker before failing or returning the thread to the pool if the conductor can automatically retry.

`rootwrap_config`

Type string

Default `/etc/ironic/rootwrap.conf`

Path to the rootwrap configuration file to use for running commands as root.

`tempdir`

Type string

Default `/tmp`

This option has a sample default set, which means that its actual default value may vary from the one documented above.

Temporary working directory, default is Python temp dir.

`webserver_verify_ca`

Type string

Default `True`

Mutable This option can be changed without restarting.

CA certificates to be used for certificate verification. This can be either a Boolean value or a path to a CA_BUNDLE file.If set to True, the certificates present in the standard path are used to verify the host certificates.If set to False, the conductor will ignore verifying the SSL certificate presented

by the host.If its a path, conductor uses the specified certificate for SSL verification. If the path does not exist, the behavior is same as when this value is set to True i.e the certificates present in the standard path are used for SSL verification.Defaults to True.

webserver_connection_timeout

Type integer

Default `60`

Connection timeout when accessing remote web servers with images.

agent

manage_agent_boot

Type boolean

Default `True`

Whether Ironic will manage booting of the agent ramdisk. If set to False, you will need to configure your mechanism to allow booting the agent ramdisk.

memory_consumed_by_agent

Type integer

Default `0`

Mutable This option can be changed without restarting.

The memory size in MiB consumed by agent when it is booted on a bare metal node. This is used for checking if the image can be downloaded and deployed on the bare metal node after booting agent ramdisk. This may be set according to the memory consumed by the agent ramdisk image.

stream_raw_images

Type boolean

Default `True`

Mutable This option can be changed without restarting.

Whether the agent ramdisk should stream raw images directly onto the disk or not. By streaming raw images directly onto the disk the agent ramdisk will not spend time copying the image to a tmpfs partition (therefore consuming less memory) prior to writing it to the disk. Unless the disk where the image will be copied to is really slow, this option should be set to True. Defaults to True.

post_deploy_get_power_state_retries

Type integer

Default `6`

Number of times to retry getting power state to check if bare metal node has been powered off after a soft power off.

`post_deploy_get_power_state_retry_interval`

Type integer

Default `5`

Amount of time (in seconds) to wait between polling power state after trigger soft poweroff.

`agent_api_version`

Type string

Default `v1`

API version to use for communicating with the ramdisk agent.

`deploy_logs_collect`

Type string

Default `on_failure`

Valid Values always, on_failure, never

Mutable This option can be changed without restarting.

Whether Ironic should collect the deployment logs on deployment failure (on_failure), always or never.

Possible values

always always collect the logs

on_failure only collect logs if there is a failure

never never collect logs

`deploy_logs_storage_backend`

Type string

Default `local`

Valid Values local, swift

Mutable This option can be changed without restarting.

The name of the storage backend where the logs will be stored.

Possible values

local store the logs locally

swift store the logs in Object Storage service

`deploy_logs_local_path`

Type string

Default `/var/log/ironic/deploy`

Mutable This option can be changed without restarting.

The path to the directory where the logs should be stored, used when the deploy_logs_storage_backend is configured to local.

`deploy_logs_swift_container`

Type string

Default `ironic_deploy_logs_container`

Mutable This option can be changed without restarting.

The name of the Swift container to store the logs, used when the deploy_logs_storage_backend is configured to swift.

`deploy_logs_swift_days_to_expire`

Type integer

Default `30`

Mutable This option can be changed without restarting.

Number of days before a log object is marked as expired in Swift. If None, the logs will be kept forever or until manually deleted. Used when the deploy_logs_storage_backend is configured to swift.

`image_download_source`

Type string

Default `http`

Valid Values swift, http, local

Mutable This option can be changed without restarting.

Specifies whether direct deploy interface should try to use the image source directly or if ironic should cache the image on the conductor and serve it from ironics own http server.

Possible values

swift IPA ramdisk retrieves instance image from the Object Storage service.

http IPA ramdisk retrieves instance image from HTTP service served at conductor nodes.

local Same as http, but HTTP images are also cached locally, converted and served from the conductor

`command_timeout`

Type integer

Default `60`

Mutable This option can be changed without restarting.

Timeout (in seconds) for IPA commands.

`max_command_attempts`

Type integer

Default `3`

This is the maximum number of attempts that will be done for IPA commands that fails due to network problems.

`command_wait_attempts`

Type integer

Default `100`

Number of attempts to check for asynchronous commands completion before timing out.

`command_wait_interval`

Type integer

Default `6`

Number of seconds to wait for between checks for asynchronous commands completion.

`neutron_agent_poll_interval`

Type integer

Default `2`

Mutable This option can be changed without restarting.

The number of seconds Neutron agent will wait between polling for device changes. This value should be the same as CONF.AGENT.polling_interval in Neutron configuration.

`neutron_agent_max_attempts`

Type integer

Default `100`

Max number of attempts to validate a Neutron agent status before raising network error for a dead agent.

`neutron_agent_status_retry_interval`

Type integer

Default `10`

Wait time in seconds between attempts for validating Neutron agent status.

`require_tls`

Type boolean

Default `False`

Mutable This option can be changed without restarting.

If set to True, callback URLs without https:// will be rejected by the conductor.

`certificates_path`

Type string

Default `/var/lib/ironic/certificates`

Path to store auto-generated TLS certificates used to validate connections to the ramdisk.

`verify_ca`

Type string

Default `True`

Path to the TLS CA to validate connection to the ramdisk. Set to True to use the system default CA storage. Set to False to disable validation. Ignored when automatic TLS setup is used.

`api_ca_file`

Type string

Default `<None>`

Path to the TLS CA that is used to start the bare metal API. In some boot methods this file can be passed to the ramdisk.

anaconda

default_ks_template

Type string

Default `$pybasedir/drivers/modules/ks.cfg.template`

Mutable This option can be changed without restarting.

kickstart template to use when no kickstart template is specified in the instance_info or the glance OS image.

ansible

ansible_extra_args

Type string

Default <None>

Extra arguments to pass on every invocation of Ansible.

verbosity

Type integer

Default <None>

Minimum Value 0

Maximum Value 4

Set ansible verbosity level requested when invoking ansible-playbook command. 4 includes detailed SSH session logging. Default is 4 when global debug is enabled and 0 otherwise.

ansible_playbook_script

Type string

Default `ansible-playbook`

Path to ansible-playbook script. Default will search the $PATH configured for user running ironic-conductor process. Provide the full path when ansible-playbook is not in $PATH or installed in not default location.

playbooks_path

Type string

Default `$pybasedir/drivers/modules/ansible/playbooks`

Path to directory with playbooks, roles and local inventory.

config_file_path

Type string

Default `$pybasedir/drivers/modules/ansible/playbooks/ansible.cfg`

Path to ansible configuration file. If set to empty, system default will be used.

`post_deploy_get_power_state_retries`

Type integer

Default `6`

Minimum Value 0

Number of times to retry getting power state to check if bare metal node has been powered off after a soft power off. Value of 0 means do not retry on failure.

`post_deploy_get_power_state_retry_interval`

Type integer

Default `5`

Minimum Value 0

Amount of time (in seconds) to wait between polling power state after trigger soft poweroff.

`extra_memory`

Type integer

Default `10`

Extra amount of memory in MiB expected to be consumed by Ansible-related processes on the node. Affects decision whether image will fit into RAM.

`image_store_insecure`

Type boolean

Default `False`

Skip verifying SSL connections to the image store when downloading the image. Setting it to True is only recommended for testing environments that use self-signed certificates.

`image_store_cafile`

Type string

Default `<None>`

Specific CA bundle to use for validating SSL connections to the image store. If not specified, CA available in the ramdisk will be used. Is not used by default playbooks included with the driver. Suitable for environments that use self-signed certificates.

`image_store_certfile`

Type string

Default `<None>`

Client cert to use for SSL connections to image store. Is not used by default playbooks included with the driver.

image_store_keyfile

Type string

Default <None>

Client key to use for SSL connections to image store. Is not used by default playbooks included with the driver.

default_username

Type string

Default ansible

Name of the user to use for Ansible when connecting to the ramdisk over SSH. It may be overridden by per-node ansible_username option in nodes driver_info field.

default_key_file

Type string

Default <None>

Absolute path to the private SSH key file to use by Ansible by default when connecting to the ramdisk over SSH. Default is to use default SSH keys configured for the user running the ironic-conductor service. Private keys with password must be pre-loaded into ssh-agent. It may be overridden by per-node ansible_key_file option in nodes driver_info field.

default_deploy_playbook

Type string

Default deploy.yaml

Path (relative to $playbooks_path or absolute) to the default playbook used for deployment. It may be overridden by per-node ansible_deploy_playbook option in nodes driver_info field.

default_shutdown_playbook

Type string

Default shutdown.yaml

Path (relative to $playbooks_path or absolute) to the default playbook used for graceful in-band shutdown of the node. It may be overridden by per-node ansible_shutdown_playbook option in nodes driver_info field.

default_clean_playbook

Type string

Default clean.yaml

Path (relative to $playbooks_path or absolute) to the default playbook used for node cleaning. It may be overridden by per-node ansible_clean_playbook option in nodes driver_info field.

default_clean_steps_config

Type string

Default `clean_steps.yaml`

Path (relative to $playbooks_path or absolute) to the default auxiliary cleaning steps file used during the node cleaning. It may be overridden by per-node ansible_clean_steps_config option in nodes driver_info field.

default_python_interpreter

Type string

Default <None>

Absolute path to the python interpreter on the managed machines. It may be overridden by per-node ansible_python_interpreter option in nodes driver_info field. By default, ansible uses /usr/bin/python

api

host_ip

Type host address

Default `0.0.0.0`

The IP address or hostname on which ironic-api listens.

port

Type port number

Default `6385`

Minimum Value 0

Maximum Value 65535

The TCP port on which ironic-api listens.

unix_socket

Type string

Default <None>

Unix socket to listen on. Disables host_ip and port.

unix_socket_mode

Type unknown type

Default <None>

File mode (an octal number) of the unix socket to listen on. Ignored if unix_socket is not set.

max_limit

Type integer

Default 1000

Mutable This option can be changed without restarting.

The maximum number of items returned in a single response from a collection resource.

public_endpoint

Type string

Default <None>

Mutable This option can be changed without restarting.

Public URL to use when building the links to the API resources (for example, https://ironic.rocks:6384). If None the links will be built using the requests host URL. If the API is operating behind a proxy, you will want to change this to represent the proxys URL. Defaults to None. Ignored when proxy headers parsing is enabled via [oslo_middleware]enable_proxy_headers_parsing option.

api_workers

Type integer

Default <None>

Number of workers for OpenStack Ironic API service. The default is equal to the number of CPUs available, but not more than 4. One worker is used if the CPU number cannot be detected.

enable_ssl_api

Type boolean

Default False

Enable the integrated stand-alone API to service requests via HTTPS instead of HTTP. If there is a front-end service performing HTTPS offloading from the service, this option should be False; note, you will want to enable proxy headers parsing with [oslo_middleware]enable_proxy_headers_parsing option or configure [api]public_endpoint option to set URLs in responses to the SSL terminated one.

restrict_lookup

Type boolean

Default True

Mutable This option can be changed without restarting.

Whether to restrict the lookup API to only nodes in certain states.

ramdisk_heartbeat_timeout

Type integer

Default `300`

Mutable This option can be changed without restarting.

Maximum interval (in seconds) for agent heartbeats.

network_data_schema

Type string

Default `$pybasedir/api/controllers/v1/network-data-schema.json`

Schema for network data used by this deployment.

audit

enabled

Type boolean

Default `False`

Enable auditing of API requests (for ironic-api service).

audit_map_file

Type string

Default `/etc/ironic/api_audit_map.conf`

Path to audit map file for ironic-api service. Used only when API audit is enabled.

ignore_req_list

Type string

Default `''`

Comma separated list of Ironic REST API HTTP methods to be ignored during audit logging. For example: auditing will not be done on any GET or POST requests if this is set to GET,POST. It is used only when API audit is enabled.

cinder

action_retries

Type integer

Default 3

Number of retries in the case of a failed action (currently only used when detaching volumes).

action_retry_interval

Type integer

Default 5

Retry interval in seconds in the case of a failed action (only specific actions are retried).

auth_url

Type unknown type

Default <None>

Authentication URL

auth_type

Type unknown type

Default <None>

Authentication type to load

Table 6: Deprecated Variations

Group	Name
cinder	auth_plugin

cafile

Type string

Default <None>

PEM encoded Certificate Authority to use when verifying HTTPs connections.

certfile

Type string

Default <None>

PEM encoded client certificate cert file

collect_timing

Type boolean

Default `False`

Collect per-API call timing information.

connect_retries

Type integer

Default `<None>`

The maximum number of retries that should be attempted for connection errors.

connect_retry_delay

Type floating point

Default `<None>`

Delay (in seconds) between two retries for connection errors. If not set, exponential retry starting with 0.5 seconds up to a maximum of 60 seconds is used.

default_domain_id

Type unknown type

Default `<None>`

Optional domain ID to use with v3 and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

default_domain_name

Type unknown type

Default `<None>`

Optional domain name to use with v3 API and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

domain_id

Type unknown type

Default `<None>`

Domain ID to scope to

domain_name

Type unknown type

Default `<None>`

Domain name to scope to

endpoint_override

Type string

Default <None>

Always use this endpoint URL for requests for this client. NOTE: The unversioned endpoint should be specified here; to request a particular API version, use the *version*, *min-version*, and/or *max-version* options.

insecure

Type boolean

Default False

Verify HTTPS connections.

keyfile

Type string

Default <None>

PEM encoded client certificate key file

max_version

Type string

Default <None>

The maximum major version of a given API, intended to be used as the upper bound of a range with min_version. Mutually exclusive with version.

min_version

Type string

Default <None>

The minimum major version of a given API, intended to be used as the lower bound of a range with max_version. Mutually exclusive with version. If min_version is given with no max_version it is as if max version is latest.

password

Type unknown type

Default <None>

Users password

project_domain_id

Type unknown type

Default <None>

Domain ID containing project

project_domain_name

Type unknown type

Default <None>

Domain name containing project

project_id

Type unknown type

Default <None>

Project ID to scope to

Table 7: Deprecated Variations

Group	Name
cinder	tenant-id
cinder	tenant_id

project_name

Type unknown type

Default <None>

Project name to scope to

Table 8: Deprecated Variations

Group	Name
cinder	tenant-name
cinder	tenant_name

region_name

Type string

Default <None>

The default region_name for endpoint URL discovery.

retries

Type integer

Default 3

Client retries in the case of a failed request connection.

service_name

Type string

Default <None>

The default service_name for endpoint URL discovery.

`service_type`

Type string

Default volumev3

The default service_type for endpoint URL discovery.

`split_loggers`

Type boolean

Default False

Log requests to multiple loggers.

`status_code_retries`

Type integer

Default <None>

The maximum number of retries that should be attempted for retriable HTTP status codes.

`status_code_retry_delay`

Type floating point

Default <None>

Delay (in seconds) between two retries for retriable status codes. If not set, exponential retry starting with 0.5 seconds up to a maximum of 60 seconds is used.

`system_scope`

Type unknown type

Default <None>

Scope for system operations

`tenant_id`

Type unknown type

Default <None>

Tenant ID

`tenant_name`

Type unknown type

Default <None>

Tenant Name

timeout

Type integer

Default <None>

Timeout value for http requests

trust_id

Type unknown type

Default <None>

Trust ID

user_domain_id

Type unknown type

Default <None>

Users domain id

user_domain_name

Type unknown type

Default <None>

Users domain name

user_id

Type unknown type

Default <None>

User id

username

Type unknown type

Default <None>

Username

Table 9: Deprecated Variations

Group	Name
cinder	user-name
cinder	user_name

valid_interfaces

Type list

Default `['internal', 'public']`

List of interfaces, in order of preference, for endpoint URL.

`version`

Type string

Default `<None>`

Minimum Major API version within a given Major API version for endpoint URL discovery. Mutually exclusive with min_version and max_version

conductor

`workers_pool_size`

Type integer

Default `100`

Minimum Value 3

The size of the workers greenthread pool. Note that 2 threads will be reserved by the conductor itself for handling heart beats and periodic tasks. On top of that, *sync_power_state_workers* will take up to 7 green threads with the default value of 8.

`heartbeat_interval`

Type integer

Default `10`

Seconds between conductor heart beats.

`heartbeat_timeout`

Type integer

Default `60`

Maximum Value 315576000

Mutable This option can be changed without restarting.

Maximum time (in seconds) since the last check-in of a conductor. A conductor is considered inactive when this time has been exceeded.

`sync_power_state_interval`

Type integer

Default `60`

Interval between syncing the node power state to the database, in seconds. Set to 0 to disable syncing.

check_provision_state_interval

Type integer

Default 60

Minimum Value 0

Interval between checks of provision timeouts, in seconds. Set to 0 to disable checks.

check_rescue_state_interval

Type integer

Default 60

Minimum Value 1

Interval (seconds) between checks of rescue timeouts.

check_allocations_interval

Type integer

Default 60

Minimum Value 0

Interval between checks of orphaned allocations, in seconds. Set to 0 to disable checks.

cache_clean_up_interval

Type integer

Default 3600

Minimum Value 0

Interval between cleaning up image caches, in seconds. Set to 0 to disable periodic clean-up.

deploy_callback_timeout

Type integer

Default 1800

Minimum Value 0

Timeout (seconds) to wait for a callback from a deploy ramdisk. Set to 0 to disable timeout.

force_power_state_during_sync

Type boolean

Default True

Mutable This option can be changed without restarting.

During sync_power_state, should the hardware power state be set to the state recorded in the database (True) or should the database be updated based on the hardware state (False).

`power_state_sync_max_retries`

Type integer

Default 3

During sync_power_state failures, limit the number of times Ironic should try syncing the hardware node power state with the node power state in DB

`sync_power_state_workers`

Type integer

Default 8

Minimum Value 1

The maximum number of worker threads that can be started simultaneously to sync nodes power states from the periodic task.

`periodic_max_workers`

Type integer

Default 8

Maximum number of worker threads that can be started simultaneously by a periodic task. Should be less than RPC thread pool size.

`node_locked_retry_attempts`

Type integer

Default 3

Number of attempts to grab a node lock.

`node_locked_retry_interval`

Type integer

Default 1

Seconds to sleep between node lock attempts.

`send_sensor_data`

Type boolean

Default False

Enable sending sensor data message via the notification bus

`send_sensor_data_interval`

Type integer

Default `600`

Minimum Value 1

Seconds between conductor sending sensor data message to ceilometer via the notification bus.

`send_sensor_data_workers`

Type integer

Default `4`

Minimum Value 1

The maximum number of workers that can be started simultaneously for send data from sensors periodic task.

`send_sensor_data_wait_timeout`

Type integer

Default `300`

The time in seconds to wait for send sensors data periodic task to be finished before allowing periodic call to happen again. Should be less than send_sensor_data_interval value.

`send_sensor_data_types`

Type list

Default `['ALL']`

List of comma separated meter types which need to be sent to Ceilometer. The default value, ALL, is a special value meaning send all the sensor data.

`send_sensor_data_for_undeployed_nodes`

Type boolean

Default `False`

The default for sensor data collection is to only collect data for machines that are deployed, however operators may desire to know if there are failures in hardware that is not presently in use. When set to true, the conductor will collect sensor information from all nodes when sensor data collection is enabled via the send_sensor_data setting.

`sync_local_state_interval`

Type integer

Default `180`

When conductors join or leave the cluster, existing conductors may need to update any persistent local state as nodes are moved around the cluster. This option controls how often, in seconds, each conductor will check for nodes that it should take over. Set it to 0 (or a negative value) to disable the check entirely.

`configdrive_swift_container`

Type string

Default `ironic_configdrive_container`

Name of the Swift container to store config drive data. Used when configdrive_use_object_store is True.

`configdrive_swift_temp_url_duration`

Type integer

Default `<None>`

Minimum Value 60

The timeout (in seconds) after which a configdrive temporary URL becomes invalid. Defaults to deploy_callback_timeout if it is set, otherwise to 1800 seconds. Used when configdrive_use_object_store is True.

`inspect_wait_timeout`

Type integer

Default `1800`

Minimum Value 0

Timeout (seconds) for waiting for node inspection. 0 - unlimited.

`automated_clean`

Type boolean

Default `True`

Mutable This option can be changed without restarting.

Enables or disables automated cleaning. Automated cleaning is a configurable set of steps, such as erasing disk drives, that are performed on the node to ensure it is in a baseline state and ready to be deployed to. This is done after instance deletion as well as during the transition from a manageable to available state. When enabled, the particular steps performed to clean a node depend on which driver that node is managed by; see the individual drivers documentation for details. NOTE: The introduction of the cleaning operation causes instance deletion to take significantly longer. In an environment where all tenants are trusted (eg, because there is only one tenant), this option could be safely disabled.

`allow_provisioning_in_maintenance`

Type boolean

Default `True`

Mutable This option can be changed without restarting.

Whether to allow nodes to enter or undergo deploy or cleaning when in maintenance mode. If this option is set to False, and a node enters maintenance during deploy or cleaning, the process will be aborted after the next heartbeat. Automated cleaning or making a node available will also fail. If True (the default), the process will begin and will pause after the node starts heartbeating. Moving it from maintenance will make the process continue.

`clean_callback_timeout`

Type integer

Default `1800`

Minimum Value 0

Timeout (seconds) to wait for a callback from the ramdisk doing the cleaning. If the timeout is reached the node will be put in the clean failed provision state. Set to 0 to disable timeout.

`rescue_callback_timeout`

Type integer

Default `1800`

Minimum Value 0

Timeout (seconds) to wait for a callback from the rescue ramdisk. If the timeout is reached the node will be put in the rescue failed provision state. Set to 0 to disable timeout.

`soft_power_off_timeout`

Type integer

Default `600`

Minimum Value 1

Mutable This option can be changed without restarting.

Timeout (in seconds) of soft reboot and soft power off operation. This value always has to be positive.

`power_state_change_timeout`

Type integer

Default `60`

Minimum Value 2

Mutable This option can be changed without restarting.

Number of seconds to wait for power operations to complete, i.e., so that a baremetal node is in the desired power state. If timed out, the power operation is considered a failure.

`power_failure_recovery_interval`

Type integer

Default `300`

Minimum Value 0

Interval (in seconds) between checking the power state for nodes previously put into maintenance mode due to power synchronization failure. A node is automatically moved out of maintenance mode once its power state is retrieved successfully. Set to 0 to disable this check.

`conductor_group`

Type string

Default `''`

Name of the conductor group to join. Can be up to 255 characters and is case insensitive. This conductor will only manage nodes with a matching conductor_group field set on the node.

`allow_deleting_available_nodes`

Type boolean

Default `True`

Mutable This option can be changed without restarting.

Allow deleting nodes which are in state available. Defaults to True.

`enable_mdns`

Type boolean

Default `False`

Whether to enable publishing the baremetal API endpoint via multicast DNS.

`deploy_kernel`

Type string

Default `<None>`

Mutable This option can be changed without restarting.

Glance ID, http:// or file:// URL of the kernel of the default deploy image.

`deploy_ramdisk`

Type string

Default `<None>`

Mutable This option can be changed without restarting.

Glance ID, http:// or file:// URL of the initramfs of the default deploy image.

`rescue_kernel`

Type string

Default `<None>`

Mutable This option can be changed without restarting.

Glance ID, http:// or file:// URL of the kernel of the default rescue image.

`rescue_ramdisk`

Type string

Default `<None>`

Mutable This option can be changed without restarting.

Glance ID, http:// or file:// URL of the initramfs of the default rescue image.

`rescue_password_hash_algorithm`

Type string

Default `sha256`

Valid Values sha256, sha512

Mutable This option can be changed without restarting.

Password hash algorithm to be used for the rescue password.

`require_rescue_password_hashed`

Type boolean

Default `False`

Mutable This option can be changed without restarting.

Option to cause the conductor to not fallback to an un-hashed version of the rescue password, permitting rescue with older ironic-python-agent ramdisks.

`bootloader`

Type string

Default `<None>`

Mutable This option can be changed without restarting.

Glance ID, http:// or file:// URL of the EFI system partition image containing EFI boot loader. This image will be used by ironic when building UEFI-bootable ISO out of kernel and ramdisk. Required for UEFI boot from partition images.

`clean_step_priority_override`

Type unknown type

Default `{}`

Priority to run automated clean steps for both in-band and out of band clean steps, provided in interface.step_name:priority format, e.g. deploy.erase_devices_metadata:123. The option can be specified multiple times to define priorities for multiple steps. If set to 0, this specific step will not run during cleaning. If unset for an inband clean step, will use the priority set in the ramdisk.

`node_history`

Type boolean

Default `True`

Mutable This option can be changed without restarting.

Boolean value, default True, if node event history is to be recorded. Errors and other noteworthy events in relation to a node are journaled to a database table which incurs some additional load. A periodic task does periodically remove entries from the database. Please note, if this is disabled, the conductor will continue to purge entries as long as [conductor]node_history_cleanup_batch_count is not 0.

`node_history_max_entries`

Type integer

Default `300`

Minimum Value 0

Mutable This option can be changed without restarting.

Maximum number of history entries which will be stored in the database per node. Default is 300. This setting excludes the minimum number of days retained using the [conductor]node_history_minimum_days setting.

`node_history_cleanup_interval`

Type integer

Default `86400`

Minimum Value 0

Interval in seconds at which node history entries can be cleaned up in the database. Setting to 0 disables the periodic task. Defaults to once a day, or 86400 seconds.

`node_history_cleanup_batch_count`

Type integer

Default `1000`

Minimum Value 0

The target number of node history records to purge from the database when performing clean-up. Deletes are performed by node, and a node with excess records for a node will still be deleted. Defaults to 1000. Operators who find node history building up may wish to lower this threshold and decrease the time between cleanup operations using the `node_history_cleanup_interval` setting.

`node_history_minimum_days`

Type integer

Default `0`

Minimum Value 0

Mutable This option can be changed without restarting.

The minimum number of days to explicitly keep on hand in the database history entries for nodes. This is exclusive from the [conductor]node_history_max_entries setting as users of this setting are anticipated to need to retain history by policy.

`verify_step_priority_override`

Type unknown type

Default `{}`

Mutable This option can be changed without restarting.

Priority to run automated verify steps provided in interface.step_name:priority format,e.g. management.clear_job_queue:123. The option can be specified multiple times to define priorities for multiple steps. If set to 0, this specific step will not run during verification.

console

`terminal`

Type string

Default `shellinaboxd`

Path to serial console terminal program. Used only by Shell In A Box console.

`terminal_cert_dir`

Type string

Default `<None>`

Directory containing the terminal SSL cert (PEM) for serial console access. Used only by Shell In A Box console.

`terminal_pid_dir`

Type string

Default `<None>`

Directory for holding terminal pid files. If not specified, the temporary directory will be used.

`terminal_timeout`

Type integer

Default `600`

Minimum Value 0

Timeout (in seconds) for the terminal session to be closed on inactivity. Set to 0 to disable timeout. Used only by Socat console.

subprocess_checking_interval

Type integer

Default 1

Time interval (in seconds) for checking the status of console subprocess.

subprocess_timeout

Type integer

Default 10

Time (in seconds) to wait for the console subprocess to start.

kill_timeout

Type integer

Default 1

Time (in seconds) to wait for the console subprocess to exit before sending SIGKILL signal.

socat_address

Type ip address

Default $my_ip

IP address of Socat service running on the host of ironic conductor. Used only by Socat console.

port_range

Type string

Default 10000:20000

This option has a sample default set, which means that its actual default value may vary from the one documented above.

A range of ports available to be used for the console proxy service running on the host of ironic conductor, in the form of <start>:<stop>. This option is used by both Shellinabox and Socat console

cors

allowed_origin

Type list

Default <None>

Indicate whether this resource may be shared with the domain received in the requests origin header. Format: <protocol>://<host>[:<port>], no trailing slash. Example: https://horizon.example.com

allow_credentials

Type boolean

Default `True`

Indicate that the actual request can include user credentials

`expose_headers`

Type list

Default `[]`

Indicate which headers are safe to expose to the API. Defaults to HTTP Simple Headers.

`max_age`

Type integer

Default `3600`

Maximum cache age of CORS preflight requests.

`allow_methods`

Type list

Default `['OPTIONS', 'GET', 'HEAD', 'POST', 'PUT', 'DELETE', 'TRACE', 'PATCH']`

Indicate which methods can be used during the actual request.

`allow_headers`

Type list

Default `[]`

Indicate which header field names may be used during the actual request.

database

`sqlite_synchronous`

Type boolean

Default `True`

If True, SQLite uses synchronous mode.

Table 10: Deprecated Variations

Group	Name
DEFAULT	sqlite_synchronous

`backend`

Type string

Default `sqlalchemy`

The back end to use for the database.

Table 11: Deprecated Variations

Group	Name
DEFAULT	db_backend

`connection`

Type string

Default `<None>`

The SQLAlchemy connection string to use to connect to the database.

Table 12: Deprecated Variations

Group	Name
DEFAULT	sql_connection
DATABASE	sql_connection
sql	connection

`slave_connection`

Type string

Default `<None>`

The SQLAlchemy connection string to use to connect to the slave database.

`mysql_sql_mode`

Type string

Default `TRADITIONAL`

The SQL mode to be used for MySQL sessions. This option, including the default, overrides any server-set SQL mode. To use whatever SQL mode is set by the server configuration, set this to no value. Example: mysql_sql_mode=

`mysql_enable_ndb`

Type boolean

Default `False`

If True, transparently enables support for handling MySQL Cluster (NDB).

`connection_recycle_time`

Type integer

Default `3600`

Connections which have been present in the connection pool longer than this number of seconds will be replaced with a new one the next time they are checked out from the pool.

`max_pool_size`

Type integer

Default `5`

Maximum number of SQL connections to keep open in a pool. Setting a value of 0 indicates no limit.

`max_retries`

Type integer

Default `10`

Maximum number of database connection retries during startup. Set to -1 to specify an infinite retry count.

Table 13: Deprecated Variations

Group	Name
DEFAULT	sql_max_retries
DATABASE	sql_max_retries

`retry_interval`

Type integer

Default `10`

Interval between retries of opening a SQL connection.

Table 14: Deprecated Variations

Group	Name
DEFAULT	sql_retry_interval
DATABASE	reconnect_interval

`max_overflow`

Type integer

Default `50`

If set, use this value for max_overflow with SQLAlchemy.

Table 15: Deprecated Variations

Group	Name
DEFAULT	sql_max_overflow
DATABASE	sqlalchemy_max_overflow

`connection_debug`

Type integer

Default `0`

Minimum Value 0

Maximum Value 100

Verbosity of SQL debugging information: 0=None, 100=Everything.

Table 16: Deprecated Variations

Group	Name
DEFAULT	sql_connection_debug

`connection_trace`

Type boolean

Default `False`

Add Python stack traces to SQL as comment strings.

Table 17: Deprecated Variations

Group	Name
DEFAULT	sql_connection_trace

`pool_timeout`

Type integer

Default `<None>`

If set, use this value for pool_timeout with SQLAlchemy.

Table 18: Deprecated Variations

Group	Name
DATABASE	sqlalchemy_pool_timeout

`use_db_reconnect`

Type boolean

Default `False`

Enable the experimental use of database reconnect on connection lost.

`db_retry_interval`

Type integer

Default `1`

Seconds between retries of a database transaction.

db_inc_retry_interval

Type boolean

Default `True`

If True, increases the interval between retries of a database operation up to db_max_retry_interval.

db_max_retry_interval

Type integer

Default `10`

If db_inc_retry_interval is set, the maximum seconds between retries of a database operation.

db_max_retries

Type integer

Default `20`

Maximum retries in case of connection error or deadlock error before error is raised. Set to -1 to specify an infinite retry count.

connection_parameters

Type string

Default `''`

Optional URL parameters to append onto the connection URL at connect time; specify as param1=value1&param2=value2&

mysql_engine

Type string

Default `InnoDB`

MySQL engine to use.

deploy

http_url

Type string

Default `<None>`

ironic-conductor nodes HTTP server URL. Example: http://192.1.2.3:8080

http_root

Type string

Default `/httpboot`

ironic-conductor nodes HTTP root path.

`external_http_url`

Type string

Default `<None>`

URL of the ironic-conductor nodes HTTP server for boot methods such as virtual media, where images could be served outside of the provisioning network. Does not apply when Swift is used. Defaults to http_url.

`external_callback_url`

Type string

Default `<None>`

Agent callback URL of the bare metal API for boot methods such as virtual media, where images could be served outside of the provisioning network. Defaults to the configuration from [service_catalog].

`enable_ata_secure_erase`

Type boolean

Default `True`

Mutable This option can be changed without restarting.

Whether to support the use of ATA Secure Erase during the cleaning process. Defaults to True.

`enable_nvme_secure_erase`

Type boolean

Default `True`

Mutable This option can be changed without restarting.

Whether to support the use of NVMe Secure Erase during the cleaning process. Currently nvme-cli format command is supported with user-data and crypto modes, depending on device capabilities.Defaults to True.

`erase_devices_priority`

Type integer

Default `<None>`

Mutable This option can be changed without restarting.

Priority to run in-band erase devices via the Ironic Python Agent ramdisk. If unset, will use the priority set in the ramdisk (defaults to 10 for the GenericHardwareManager). If set to 0, will not run during cleaning.

`erase_devices_metadata_priority`

Type integer

Default <None>

Mutable This option can be changed without restarting.

Priority to run in-band clean step that erases metadata from devices, via the Ironic Python Agent ramdisk. If unset, will use the priority set in the ramdisk (defaults to 99 for the GenericHardwareManager). If set to 0, will not run during cleaning.

`delete_configuration_priority`

Type integer

Default <None>

Mutable This option can be changed without restarting.

Priority to run in-band clean step that erases RAID configuration from devices, via the Ironic Python Agent ramdisk. If unset, will use the priority set in the ramdisk (defaults to 0 for the GenericHardwareManager). If set to 0, will not run during cleaning.

`create_configuration_priority`

Type integer

Default <None>

Mutable This option can be changed without restarting.

Priority to run in-band clean step that creates RAID configuration from devices, via the Ironic Python Agent ramdisk. If unset, will use the priority set in the ramdisk (defaults to 0 for the GenericHardwareManager). If set to 0, will not run during cleaning.

`shred_random_overwrite_iterations`

Type integer

Default 1

Minimum Value 0

Mutable This option can be changed without restarting.

During shred, overwrite all block devices N times with random data. This is only used if a device could not be ATA Secure Erased. Defaults to 1.

`shred_final_overwrite_with_zeros`

Type boolean

Default True

Mutable This option can be changed without restarting.

Whether to write zeros to a nodes block devices after writing random data. This will write zeros to the device even when deploy.shred_random_overwrite_iterations is 0. This option is only used if a device could not be ATA Secure Erased. Defaults to True.

`continue_if_disk_secure_erase_fails`

Type boolean

Default `False`

Mutable This option can be changed without restarting.

Defines what to do if a secure erase operation (NVMe or ATA) fails during cleaning in the Ironic Python Agent. If False, the cleaning operation will fail and the node will be put in `clean failed` state. If True, shred will be invoked and cleaning will continue.

`disk_erasure_concurrency`

Type integer

Default 1

Minimum Value 1

Mutable This option can be changed without restarting.

Defines the target pool size used by Ironic Python Agent ramdisk to erase disk devices. The number of threads created to erase disks will not exceed this value or the number of disks to be erased.

`power_off_after_deploy_failure`

Type boolean

Default `True`

Mutable This option can be changed without restarting.

Whether to power off a node after deploy failure. Defaults to True.

`default_boot_option`

Type string

Default `local`

Valid Values netboot, local

Mutable This option can be changed without restarting.

Default boot option to use when no boot option is requested in nodes driver_info. Defaults to local. Prior to the Ussuri release, the default was netboot.

Possible values

netboot boot from a network

local local boot

> **Warning:** This option is deprecated for removal. Its value may be silently ignored in the future.
>
> **Reason** Support for network boot will be removed after the Yoga release.

`default_boot_mode`

Type string

Default `uefi`

Valid Values uefi, bios

Mutable This option can be changed without restarting.

Default boot mode to use when no boot mode is requested in nodes driver_info, capabilities or in the *instance_info* configuration. Currently the default boot mode is uefi, but it was bios previously in Ironic. It is recommended to set an explicit value for this option, and if the setting or default differs from nodes, to ensure that nodes are configured specifically for their desired boot mode. This option only has effect when management interface supports boot mode management

Possible values

uefi UEFI boot mode

bios Legacy BIOS boot mode

`configdrive_use_object_store`

Type boolean

Default `False`

Mutable This option can be changed without restarting.

Whether to upload the config drive to object store. Set this option to True to store config drive in a swift endpoint.

Table 19: Deprecated Variations

Group	Name
conductor	configdrive_use_swift

`http_image_subdir`

Type string

Default `agent_images`

The name of subdirectory under ironic-conductor nodes HTTP root path which is used to place instance images for the direct deploy interface, when local HTTP service is incorporated to provide instance image instead of swift tempurls.

`fast_track`

Type boolean

Default `False`

Mutable This option can be changed without restarting.

Whether to allow deployment agents to perform lookup, heartbeat operations during initial states of a machine lifecycle and by-pass the normal setup procedures for a ramdisk. This feature also enables power operations which are part of deployment processes to be bypassed if the ramdisk has performed a heartbeat operation using the fast_track_timeout setting.

`fast_track_timeout`

Type integer

Default `300`

Minimum Value 0

Maximum Value 300

Mutable This option can be changed without restarting.

Seconds for which the last heartbeat event is to be considered valid for the purpose of a fast track sequence. This setting should generally be less than the number of seconds for Power-On Self Test and typical ramdisk start-up. This value should not exceed the [api]ramdisk_heartbeat_timeout setting.

`erase_skip_read_only`

Type boolean

Default `False`

Mutable This option can be changed without restarting.

If the ironic-python-agent should skip read-only devices when running the erase_devices clean step where block devices are zeroed out. This requires ironic-python-agent 6.0.0 or greater. By default a read-only device will cause non-metadata based cleaning operations to fail due to the possible operational security risk of data being retained between deployments of the bare metal node.

`ramdisk_image_download_source`

Type string

Default `local`

Valid Values http, local, swift

Mutable This option can be changed without restarting.

Specifies whether a boot iso image should be served from its own original location using the image source url directly, or if ironic should cache the image on the conductor and serve it from ironics own http server.

Possible values

http In case the ramdisk is already a bootable iso, using this option it will be directly provided by an external HTTP service using its full url.

local This is the default behavior. The image is downloaded, prepared and cached locally, to be served from the conductor.

swift Same as http, but if the image is a Glance UUID, it is exposed via a Swift temporary URL.

`iso_master_path`

Type string

Default `/var/lib/ironic/master_iso_images`

On the ironic-conductor node, directory where master ISO images are stored on disk. Setting to the empty string disables image caching.

`iso_cache_size`

Type integer

Default `20480`

Maximum size (in MiB) of cache for master ISO images, including those in use.

`iso_cache_ttl`

Type integer

Default `10080`

Maximum TTL (in minutes) for old master ISO images in cache.

dhcp

`dhcp_provider`

Type string

Default `neutron`

DHCP provider to use. neutron uses Neutron, and none uses a no-op provider.

disk_partitioner

check_device_interval

Type integer

Default 1

After Ironic has completed creating the partition table, it continues to check for activity on the attached iSCSI device status at this interval prior to copying the image to the node, in seconds

check_device_max_retries

Type integer

Default 20

The maximum number of times to check that the device is not accessed by another process. If the device is still busy after that, the disk partitioning will be treated as having failed.

disk_utils

efi_system_partition_size

Type integer

Default 200

Size of EFI system partition in MiB when configuring UEFI systems for local boot.

bios_boot_partition_size

Type integer

Default 1

Size of BIOS Boot partition in MiB when configuring GPT partitioned systems for local boot in BIOS.

dd_block_size

Type string

Default 1M

Block size to use when writing to the nodes disk.

partition_detection_attempts

Type integer

Default 3

Minimum Value 1

Maximum attempts to detect a newly created partition.

partprobe_attempts

Type integer

Default 10

Maximum number of attempts to try to read the partition.

image_convert_memory_limit

Type integer

Default 2048

Memory limit for qemu-img convert in MiB. Implemented via the address space resource limit.

image_convert_attempts

Type integer

Default 3

Number of attempts to convert an image.

drac

query_raid_config_job_status_interval

Type integer

Default 120

Minimum Value 1

Interval (in seconds) between periodic RAID job status checks to determine whether the asynchronous RAID configuration was successfully finished or not.

boot_device_job_status_timeout

Type integer

Default 30

Minimum Value 1

Maximum amount of time (in seconds) to wait for the boot device configuration job to transition to the correct state to allow a reboot or power on to complete.

config_job_max_retries

Type integer

Default 240

Minimum Value 1

Maximum number of retries for the configuration job to complete successfully.

`query_import_config_job_status_interval`

Type integer

Default `60`

Minimum Value 0

Number of seconds to wait between checking for completed import configuration task

`bios_factory_reset_timeout`

Type integer

Default `600`

Minimum Value 1

Maximum time (in seconds) to wait for factory reset of BIOS settings to complete.

`raid_job_timeout`

Type integer

Default `300`

Minimum Value 1

Maximum time (in seconds) to wait for RAID job to complete

glance

`allowed_direct_url_schemes`

Type list

Default `[]`

A list of URL schemes that can be downloaded directly via the direct_url. Currently supported schemes: [file].

`auth_url`

Type unknown type

Default `<None>`

Authentication URL

`auth_type`

Type unknown type

Default `<None>`

Authentication type to load

Table 20: Deprecated Variations

Group	Name
glance	auth_plugin

cafile

Type string

Default <None>

PEM encoded Certificate Authority to use when verifying HTTPs connections.

certfile

Type string

Default <None>

PEM encoded client certificate cert file

collect_timing

Type boolean

Default False

Collect per-API call timing information.

connect_retries

Type integer

Default <None>

The maximum number of retries that should be attempted for connection errors.

connect_retry_delay

Type floating point

Default <None>

Delay (in seconds) between two retries for connection errors. If not set, exponential retry starting with 0.5 seconds up to a maximum of 60 seconds is used.

default_domain_id

Type unknown type

Default <None>

Optional domain ID to use with v3 and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

default_domain_name

Type unknown type

Default <None>

Optional domain name to use with v3 API and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

domain_id

Type unknown type

Default <None>

Domain ID to scope to

domain_name

Type unknown type

Default <None>

Domain name to scope to

endpoint_override

Type string

Default <None>

Always use this endpoint URL for requests for this client. NOTE: The unversioned endpoint should be specified here; to request a particular API version, use the *version*, *min-version*, and/or *max-version* options.

insecure

Type boolean

Default False

Verify HTTPS connections.

keyfile

Type string

Default <None>

PEM encoded client certificate key file

max_version

Type string

Default <None>

The maximum major version of a given API, intended to be used as the upper bound of a range with min_version. Mutually exclusive with version.

min_version

Type string

Default <None>

The minimum major version of a given API, intended to be used as the lower bound of a range with max_version. Mutually exclusive with version. If min_version is given with no max_version it is as if max version is latest.

num_retries

Type integer

Default 0

Number of retries when downloading an image from glance.

password

Type unknown type

Default <None>

Users password

project_domain_id

Type unknown type

Default <None>

Domain ID containing project

project_domain_name

Type unknown type

Default <None>

Domain name containing project

project_id

Type unknown type

Default <None>

Project ID to scope to

Table 21: Deprecated Variations

Group	Name
glance	tenant-id
glance	tenant_id

project_name

Type unknown type

Default <None>

Project name to scope to

Table 22: Deprecated Variations

Group	Name
glance	tenant-name
glance	tenant_name

region_name

Type string

Default <None>

The default region_name for endpoint URL discovery.

service_name

Type string

Default <None>

The default service_name for endpoint URL discovery.

service_type

Type string

Default image

The default service_type for endpoint URL discovery.

split_loggers

Type boolean

Default False

Log requests to multiple loggers.

status_code_retries

Type integer

Default <None>

The maximum number of retries that should be attempted for retriable HTTP status codes.

status_code_retry_delay

Type floating point

Default <None>

Delay (in seconds) between two retries for retriable status codes. If not set, exponential retry starting with 0.5 seconds up to a maximum of 60 seconds is used.

swift_account

Type string

Default <None>

The account that Glance uses to communicate with Swift. The format is AUTH_uuid. uuid is the UUID for the account configured in the glance-api.conf. For example: AUTH_a422b2-91f3-2f46-74b7-d7c9e8958f5d30. If not set, the default value is calculated based on the ID of the project used to access Swift (as set in the [swift] section). Swift temporary URL format: endpoint_url/api_version/account/container/object_id

swift_account_prefix

Type string

Default AUTH

The prefix added to the project uuid to determine the swift account.

swift_api_version

Type string

Default v1

The Swift API version to create a temporary URL for. Defaults to v1. Swift temporary URL format: endpoint_url/api_version/account/container/object_id

swift_container

Type string

Default glance

The Swift container Glance is configured to store its images in. Defaults to glance, which is the default in glance-api.conf. Swift temporary URL format: endpoint_url/api_version/account/container/object_id

swift_endpoint_url

Type string

Default <None>

The endpoint (scheme, hostname, optional port) for the Swift URL of the form endpoint_url/api_version/account/container/object_id. Do not include trailing /. For example, use https://swift.example.com. If using RADOS Gateway, endpoint may also contain /swift path; if it does not, it will be appended. Used for temporary URLs, will be fetched from the service catalog, if not provided.

swift_store_multiple_containers_seed

Type integer

Default `0`

This should match a config by the same name in the Glance configuration file. When set to 0, a single-tenant store will only use one container to store all images. When set to an integer value between 1 and 32, a single-tenant store will use multiple containers to store images, and this value will determine how many containers are created.

`swift_temp_url_cache_enabled`

Type boolean

Default `False`

Whether to cache generated Swift temporary URLs. Setting it to true is only useful when an image caching proxy is used. Defaults to False.

`swift_temp_url_duration`

Type integer

Default `1200`

The length of time in seconds that the temporary URL will be valid for. Defaults to 20 minutes. If some deploys get a 401 response code when trying to download from the temporary URL, try raising this duration. This value must be greater than or equal to the value for swift_temp_url_expected_download_start_delay

`swift_temp_url_expected_download_start_delay`

Type integer

Default `0`

Minimum Value 0

This is the delay (in seconds) from the time of the deploy request (when the Swift temporary URL is generated) to when the IPA ramdisk starts up and URL is used for the image download. This value is used to check if the Swift temporary URL duration is large enough to let the image download begin. Also if temporary URL caching is enabled this will determine if a cached entry will still be valid when the download starts. swift_temp_url_duration value must be greater than or equal to this options value. Defaults to 0.

`swift_temp_url_key`

Type string

Default `<None>`

The secret token given to Swift to allow temporary URL downloads. Required for temporary URLs. For the Swift backend, the key on the service project (as set in the [swift] section) is used by default.

`system_scope`

Type unknown type

Default `<None>`

Scope for system operations

tenant_id

Type unknown type

Default <None>

Tenant ID

tenant_name

Type unknown type

Default <None>

Tenant Name

timeout

Type integer

Default <None>

Timeout value for http requests

trust_id

Type unknown type

Default <None>

Trust ID

user_domain_id

Type unknown type

Default <None>

Users domain id

user_domain_name

Type unknown type

Default <None>

Users domain name

user_id

Type unknown type

Default <None>

User id

username

Type unknown type

Default <None>

Username

Table 23: Deprecated Variations

Group	Name
glance	user-name
glance	user_name

valid_interfaces

Type list

Default ['internal', 'public']

List of interfaces, in order of preference, for endpoint URL.

version

Type string

Default <None>

Minimum Major API version within a given Major API version for endpoint URL discovery. Mutually exclusive with min_version and max_version

healthcheck

path

Type string

Default /healthcheck

The path to respond to healtcheck requests on.

> **Warning:** This option is deprecated for removal. Its value may be silently ignored in the future.

detailed

Type boolean

Default False

Show more detailed information as part of the response. Security note: Enabling this option may expose sensitive details about the service being monitored. Be sure to verify that it will not violate your security policies.

backends

Type list

Default []

Additional backends that can perform health checks and report that information back as part of a request.

`disable_by_file_path`

Type string

Default <None>

Check the presence of a file to determine if an application is running on a port. Used by DisableByFileHealthcheck plugin.

`disable_by_file_paths`

Type list

Default []

Check the presence of a file based on a port to determine if an application is running on a port. Expects a port:path list of strings. Used by DisableByFilesPortsHealthcheck plugin.

`enabled`

Type boolean

Default False

Enable the health check endpoint at /healthcheck. Note that this is unauthenticated. More information is available at https://docs.openstack.org/oslo.middleware/latest/reference/healthcheck_plugins.html.

ilo

`client_timeout`

Type integer

Default 60

Timeout (in seconds) for iLO operations

`client_port`

Type port number

Default 443

Minimum Value 0

Maximum Value 65535

Port to be used for iLO operations

`swift_ilo_container`

Type string

Default `ironic_ilo_container`

The Swift iLO container to store data.

`swift_object_expiry_timeout`

Type integer

Default `900`

Amount of time in seconds for Swift objects to auto-expire.

`use_web_server_for_images`

Type boolean

Default `False`

Set this to True to use http web server to host floppy images and generated boot ISO. This requires http_root and http_url to be configured in the [deploy] section of the config file. If this is set to False, then Ironic will use Swift to host the floppy images and generated boot_iso.

`clean_priority_reset_ilo`

Type integer

Default `0`

Priority for reset_ilo clean step.

`clean_priority_reset_bios_to_default`

Type integer

Default `10`

Priority for reset_bios_to_default clean step.

`clean_priority_reset_secure_boot_keys_to_default`

Type integer

Default `20`

Priority for reset_secure_boot_keys clean step. This step will reset the secure boot keys to manufacturing defaults.

`clean_priority_clear_secure_boot_keys`

Type integer

Default `0`

Priority for clear_secure_boot_keys clean step. This step is not enabled by default. It can be enabled to clear all secure boot keys enrolled with iLO.

`clean_priority_reset_ilo_credential`

Type integer

Default `30`

Priority for reset_ilo_credential clean step. This step requires ilo_change_password parameter to be updated in nodess driver_info with the new password.

`power_wait`

Type integer

Default `2`

Amount of time in seconds to wait in between power operations

`oob_erase_devices_job_status_interval`

Type integer

Default `300`

Minimum Value 10

Interval (in seconds) between periodic erase-devices status checks to determine whether the asynchronous out-of-band erase-devices was successfully finished or not. On an average, a 300GB HDD with default pattern overwrite would take approximately 9 hours and 300GB SSD with default pattern block would take approx. 30 seconds to complete sanitize disk erase.

`ca_file`

Type string

Default `<None>`

CA certificate file to validate iLO.

> **Warning:** This option is deprecated for removal. Its value may be silently ignored in the future.
>
> **Reason** Its being replaced by new configuration parameter verify_ca.

`verify_ca`

Type string

Default `True`

CA certificate to validate iLO. This can be either a Boolean value, a path to a CA_BUNDLE file or directory with certificates of trusted CAs. If set to True the driver will verify the host certificates;

if False the driver will ignore verifying the SSL certificate. If its a path the driver will use the specified certificate or one of the certificates in the directory. Defaults to True.

`default_boot_mode`

Type string

Default `auto`

Valid Values auto, bios, uefi

Default boot mode to be used in provisioning when boot_mode capability is not provided in the properties/capabilities of the node. The default is auto for backward compatibility. When auto is specified, default boot mode will be selected based on boot mode settings on the system.

Possible values

auto based on boot mode settings on the system

bios BIOS boot mode

uefi UEFI boot mode

`file_permission`

Type integer

Default `420`

File permission for swift-less image hosting with the octal permission representation of file access permissions. This setting defaults to `644`, or as the octal number `0o644` in Python. This setting must be set to the octal number representation, meaning starting with `0o`.

`kernel_append_params`

Type string

Default `nofb nomodeset vga=normal`

Mutable This option can be changed without restarting.

Additional kernel parameters to pass down to the instance kernel. These parameters can be consumed by the kernel or by the applications by reading /proc/cmdline. Mind severe cmdline size limit! Can be overridden by *instance_info/kernel_append_params* property.

inspector

`auth_url`

Type unknown type

Default `<None>`

Authentication URL

`auth_type`

Type unknown type

Default <None>

Authentication type to load

Table 24: Deprecated Variations

Group	Name
inspector	auth_plugin

`cafile`

Type string

Default <None>

PEM encoded Certificate Authority to use when verifying HTTPs connections.

`callback_endpoint_override`

Type string

Default <None>

endpoint to use as a callback for posting back introspection data when boot is managed by ironic. Standard keystoneauth options are used by default.

`certfile`

Type string

Default <None>

PEM encoded client certificate cert file

`collect_timing`

Type boolean

Default `False`

Collect per-API call timing information.

`connect_retries`

Type integer

Default <None>

The maximum number of retries that should be attempted for connection errors.

`connect_retry_delay`

Type floating point

Default <None>

Delay (in seconds) between two retries for connection errors. If not set, exponential retry starting with 0.5 seconds up to a maximum of 60 seconds is used.

`default_domain_id`

Type unknown type

Default <None>

Optional domain ID to use with v3 and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

`default_domain_name`

Type unknown type

Default <None>

Optional domain name to use with v3 API and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

`domain_id`

Type unknown type

Default <None>

Domain ID to scope to

`domain_name`

Type unknown type

Default <None>

Domain name to scope to

`endpoint_override`

Type string

Default <None>

Always use this endpoint URL for requests for this client. NOTE: The unversioned endpoint should be specified here; to request a particular API version, use the *version*, *min-version*, and/or *max-version* options.

`extra_kernel_params`

Type string

Default ''

extra kernel parameters to pass to the inspection ramdisk when boot is managed by ironic (not ironic-inspector). Pairs key=value separated by spaces.

insecure

Type boolean

Default `False`

Verify HTTPS connections.

keyfile

Type string

Default `<None>`

PEM encoded client certificate key file

max_version

Type string

Default `<None>`

The maximum major version of a given API, intended to be used as the upper bound of a range with min_version. Mutually exclusive with version.

min_version

Type string

Default `<None>`

The minimum major version of a given API, intended to be used as the lower bound of a range with max_version. Mutually exclusive with version. If min_version is given with no max_version it is as if max version is latest.

password

Type unknown type

Default `<None>`

Users password

power_off

Type boolean

Default `True`

whether to power off a node after inspection finishes. Ignored for nodes that have fast track mode enabled.

project_domain_id

Type unknown type

Default `<None>`

Domain ID containing project

project_domain_name

Type unknown type

Default <None>

Domain name containing project

project_id

Type unknown type

Default <None>

Project ID to scope to

Table 25: Deprecated Variations

Group	Name
inspector	tenant-id
inspector	tenant_id

project_name

Type unknown type

Default <None>

Project name to scope to

Table 26: Deprecated Variations

Group	Name
inspector	tenant-name
inspector	tenant_name

region_name

Type string

Default <None>

The default region_name for endpoint URL discovery.

require_managed_boot

Type boolean

Default False

require that the in-band inspection boot is fully managed by ironic. Set this to True if your installation of ironic-inspector does not have a separate PXE boot environment.

service_name

Type string

Default <None>

The default service_name for endpoint URL discovery.

`service_type`

Type string

Default `baremetal-introspection`

The default service_type for endpoint URL discovery.

`split_loggers`

Type boolean

Default `False`

Log requests to multiple loggers.

`status_code_retries`

Type integer

Default <None>

The maximum number of retries that should be attempted for retriable HTTP status codes.

`status_code_retry_delay`

Type floating point

Default <None>

Delay (in seconds) between two retries for retriable status codes. If not set, exponential retry starting with 0.5 seconds up to a maximum of 60 seconds is used.

`status_check_period`

Type integer

Default `60`

period (in seconds) to check status of nodes on inspection

`system_scope`

Type unknown type

Default <None>

Scope for system operations

`tenant_id`

Type unknown type

Default <None>

Tenant ID

tenant_name

Type unknown type

Default <None>

Tenant Name

timeout

Type integer

Default <None>

Timeout value for http requests

trust_id

Type unknown type

Default <None>

Trust ID

user_domain_id

Type unknown type

Default <None>

Users domain id

user_domain_name

Type unknown type

Default <None>

Users domain name

user_id

Type unknown type

Default <None>

User id

username

Type unknown type

Default <None>

Username

Table 27: Deprecated Variations

Group	Name
inspector	user-name
inspector	user_name

valid_interfaces

Type list

Default `['internal', 'public']`

List of interfaces, in order of preference, for endpoint URL.

version

Type string

Default `<None>`

Minimum Major API version within a given Major API version for endpoint URL discovery. Mutually exclusive with min_version and max_version

ipmi

command_retry_timeout

Type integer

Default `60`

Mutable This option can be changed without restarting.

Maximum time in seconds to retry retryable IPMI operations. (An operation is retryable, for example, if the requested operation fails because the BMC is busy.) Setting this too high can cause the sync power state periodic task to hang when there are slow or unresponsive BMCs.

min_command_interval

Type integer

Default `5`

Mutable This option can be changed without restarting.

Minimum time, in seconds, between IPMI operations sent to a server. There is a risk with some hardware that setting this too low may cause the BMC to crash. Recommended setting is 5 seconds.

use_ipmitool_retries

Type boolean

Default `False`

When set to True and the parameters are supported by ipmitool, the number of retries and the retry interval are passed to ipmitool as parameters, and ipmitool will do the retries. When set to False, ironic will retry the ipmitool commands. Recommended setting is False

`kill_on_timeout`

Type boolean

Default `True`

Mutable This option can be changed without restarting.

Kill *ipmitool* process invoked by ironic to read node power state if *ipmitool* process does not exit after *command_retry_timeout* timeout expires. Recommended setting is True

`disable_boot_timeout`

Type boolean

Default `True`

Mutable This option can be changed without restarting.

Default timeout behavior whether ironic sends a raw IPMI command to disable the 60 second timeout for booting. Setting this option to False will NOT send that command, the default value is True. It may be overridden by per-node ipmi_disable_boot_timeout option in nodes driver_info field.

`additional_retryable_ipmi_errors`

Type multi-valued

Default `''`

Mutable This option can be changed without restarting.

Additional errors ipmitool may encounter, specific to the environment it is run in.

`debug`

Type boolean

Default `False`

Mutable This option can be changed without restarting.

Enables all ipmi commands to be executed with an additional debugging output. This is a separate option as ipmitool can log a substantial amount of misleading text when in this mode.

`cipher_suite_versions`

Type list

Default `[]`

List of possible cipher suites versions that can be supported by the hardware in case the field *cipher_suite* is not set for the node.

irmc

remote_image_share_root

Type string

Default /remote_image_share_root

Ironic conductor nodes NFS or CIFS root path

remote_image_server

Type string

Default <None>

IP of remote image server

remote_image_share_type

Type string

Default CIFS

Valid Values CIFS, NFS

Share type of virtual media

Possible values

CIFS CIFS (Common Internet File System) protocol

NFS NFS (Network File System) protocol

remote_image_share_name

Type string

Default share

share name of remote_image_server

remote_image_user_name

Type string

Default <None>

User name of remote_image_server

remote_image_user_password

Type string

Default <None>

Password of remote_image_user_name

`remote_image_user_domain`

Type string

Default `''`

Domain name of remote_image_user_name

`port`

Type port number

Default `443`

Minimum Value 0

Maximum Value 65535

Valid Values 443, 80

Port to be used for iRMC operations

Possible values

443 port 443

80 port 80

`auth_method`

Type string

Default `basic`

Valid Values basic, digest

Authentication method to be used for iRMC operations

Possible values

basic Basic authentication

digest Digest authentication

`client_timeout`

Type integer

Default `60`

Timeout (in seconds) for iRMC operations

`sensor_method`

Type string

Default `ipmitool`

Valid Values ipmitool, scci

Sensor data retrieval method.

Possible values

ipmitool IPMItool

scci Fujitsu SCCI (ServerView Common Command Interface)

`snmp_version`

Type string

Default `v2c`

Valid Values v1, v2c, v3

SNMP protocol version

Possible values

v1 SNMPv1

v2c SNMPv2c

v3 SNMPv3

`snmp_port`

Type port number

Default `161`

Minimum Value 0

Maximum Value 65535

SNMP port

`snmp_community`

Type string

Default `public`

SNMP community. Required for versions v1 and v2c

`snmp_security`

Type string

Default `<None>`

SNMP security name. Required for version v3. Will be ignored if driver_info/irmc_snmp_uer is set.

`snmp_polling_interval`

Type integer

Default `10`

SNMP polling interval in seconds

`snmp_auth_proto`

Type string

Default `sha`

Valid Values sha, sha256, sha384, sha512

SNMPv3 message authentication protocol ID. Required for version v3. Will be ignored if the version of python-scciclient is before 0.12.2. The valid options are sha, sha256, sha384 and sha512, while sha is the only supported protocol in iRMC S4 and S5, and from iRMC S6, sha256, sha384 and sha512 are supported, but sha is not supported any more.

Possible values

sha Secure Hash Algorithm 1, supported in iRMC S4 and S5.

sha256 Secure Hash Algorithm 2 with 256 bits digest, only supported in iRMC S6.

sha384 Secure Hash Algorithm 2 with 384 bits digest, only supported in iRMC S6.

sha512 Secure Hash Algorithm 2 with 512 bits digest, only supported in iRMC S6.

`snmp_priv_proto`

Type string

Default `aes`

Valid Values aes

SNMPv3 message privacy (encryption) protocol ID. Required for version v3. Will be ignored if the version of python-scciclient is before 0.12.2. aes is supported.

Possible values

aes Advanced Encryption Standard

`clean_priority_restore_irmc_bios_config`

Type integer

Default `0`

Priority for restore_irmc_bios_config clean step.

gpu_ids

Type list

Default []

List of vendor IDs and device IDs for GPU device to inspect. List items are in format vendorID/deviceID and separated by commas. GPU inspection will use this value to count the number of GPU device in a node. If this option is not defined, then leave out pci_gpu_devices in capabilities property. Sample gpu_ids value: 0x1000/0x0079,0x2100/0x0080

fpga_ids

Type list

Default []

List of vendor IDs and device IDs for CPU FPGA to inspect. List items are in format vendorID/deviceID and separated by commas. CPU inspection will use this value to find existence of CPU FPGA in a node. If this option is not defined, then leave out CUSTOM_CPU_FPGA in node traits. Sample fpga_ids value: 0x1000/0x0079,0x2100/0x0080

query_raid_config_fgi_status_interval

Type integer

Default 300

Minimum Value 1

Interval (in seconds) between periodic RAID status checks to determine whether the asynchronous RAID configuration was successfully finished or not. Foreground Initialization (FGI) will start 5 minutes after creating virtual drives.

kernel_append_params

Type string

Default <None>

Mutable This option can be changed without restarting.

Additional kernel parameters to pass down to the instance kernel. These parameters can be consumed by the kernel or by the applications by reading /proc/cmdline. Mind severe cmdline size limit! Can be overridden by *instance_info/kernel_append_params* property.

ironic_lib

fatal_exception_format_errors

Type boolean

Default `False`

Used if there is a formatting error when generating an exception message (a programming error). If True, raise an exception; if False, use the unformatted message.

Table 28: Deprecated Variations

Group	Name
DEFAULT	fatal_exception_format_errors

root_helper

Type string

Default `sudo ironic-rootwrap /etc/ironic/rootwrap.conf`

Command that is prefixed to commands that are run as root. If not specified, no commands are run as root.

json_rpc

auth_strategy

Type string

Default `<None>`

Valid Values noauth, keystone, http_basic

Authentication strategy used by JSON RPC. Defaults to the global auth_strategy setting.

Possible values

noauth no authentication

keystone use the Identity service for authentication

http_basic HTTP basic authentication

http_basic_auth_user_file

Type string

Default `/etc/ironic/htpasswd-json-rpc`

Path to Apache format user authentication file used when auth_strategy=http_basic

host_ip

Type host address

Default ::

The IP address or hostname on which JSON RPC will listen.

`port`

Type port number

Default 8089

Minimum Value 0

Maximum Value 65535

The port to use for JSON RPC

`use_ssl`

Type boolean

Default False

Whether to use TLS for JSON RPC

`http_basic_username`

Type string

Default <None>

Name of the user to use for HTTP Basic authentication client requests.

> **Warning:** This option is deprecated for removal. Its value may be silently ignored in the future.
>
> **Reason** Use username instead

`http_basic_password`

Type string

Default <None>

Password to use for HTTP Basic authentication client requests.

> **Warning:** This option is deprecated for removal. Its value may be silently ignored in the future.
>
> **Reason** Use password instead

`allowed_roles`

Type list

Default `['admin']`

List of roles allowed to use JSON RPC

`auth_url`

Type unknown type

Default `<None>`

Authentication URL

`auth_type`

Type unknown type

Default `<None>`

Authentication type to load

Table 29: Deprecated Variations

Group	Name
json_rpc	auth_plugin

`cafile`

Type string

Default `<None>`

PEM encoded Certificate Authority to use when verifying HTTPs connections.

`certfile`

Type string

Default `<None>`

PEM encoded client certificate cert file

`collect_timing`

Type boolean

Default `False`

Collect per-API call timing information.

`default_domain_id`

Type unknown type

Default `<None>`

Optional domain ID to use with v3 and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

default_domain_name

Type unknown type

Default <None>

Optional domain name to use with v3 API and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

domain_id

Type unknown type

Default <None>

Domain ID to scope to

domain_name

Type unknown type

Default <None>

Domain name to scope to

insecure

Type boolean

Default False

Verify HTTPS connections.

keyfile

Type string

Default <None>

PEM encoded client certificate key file

password

Type unknown type

Default <None>

Users password

project_domain_id

Type unknown type

Default <None>

Domain ID containing project

project_domain_name

Type unknown type

Default <None>

Domain name containing project

`project_id`

Type unknown type

Default <None>

Project ID to scope to

Table 30: Deprecated Variations

Group	Name
json_rpc	tenant-id
json_rpc	tenant_id

`project_name`

Type unknown type

Default <None>

Project name to scope to

Table 31: Deprecated Variations

Group	Name
json_rpc	tenant-name
json_rpc	tenant_name

`split_loggers`

Type boolean

Default `False`

Log requests to multiple loggers.

`system_scope`

Type unknown type

Default <None>

Scope for system operations

`tenant_id`

Type unknown type

Default <None>

Tenant ID

tenant_name

Type unknown type

Default <None>

Tenant Name

timeout

Type integer

Default <None>

Timeout value for http requests

trust_id

Type unknown type

Default <None>

Trust ID

user_domain_id

Type unknown type

Default <None>

Users domain id

user_domain_name

Type unknown type

Default <None>

Users domain name

user_id

Type unknown type

Default <None>

User id

username

Type unknown type

Default <None>

Username

Table 32: Deprecated Variations

Group	Name
json_rpc	user-name
json_rpc	user_name

keystone_authtoken

www_authenticate_uri

Type string

Default <None>

Complete public Identity API endpoint. This endpoint should not be an admin endpoint, as it should be accessible by all end users. Unauthenticated clients are redirected to this endpoint to authenticate. Although this endpoint should ideally be unversioned, client support in the wild varies. If youre using a versioned v2 endpoint here, then this should *not* be the same endpoint the service user utilizes for validating tokens, because normal end users may not be able to reach that endpoint.

Table 33: Deprecated Variations

Group	Name
keystone_authtoken	auth_uri

auth_uri

Type string

Default <None>

Complete public Identity API endpoint. This endpoint should not be an admin endpoint, as it should be accessible by all end users. Unauthenticated clients are redirected to this endpoint to authenticate. Although this endpoint should ideally be unversioned, client support in the wild varies. If youre using a versioned v2 endpoint here, then this should *not* be the same endpoint the service user utilizes for validating tokens, because normal end users may not be able to reach that endpoint. This option is deprecated in favor of www_authenticate_uri and will be removed in the S release.

Warning: This option is deprecated for removal since Queens. Its value may be silently ignored in the future.

Reason The auth_uri option is deprecated in favor of www_authenticate_uri and will be removed in the S release.

auth_version

Type string

Default <None>

API version of the Identity API endpoint.

interface

Type string

Default internal

Interface to use for the Identity API endpoint. Valid values are public, internal (default) or admin.

delay_auth_decision

Type boolean

Default False

Do not handle authorization requests within the middleware, but delegate the authorization decision to downstream WSGI components.

http_connect_timeout

Type integer

Default <None>

Request timeout value for communicating with Identity API server.

http_request_max_retries

Type integer

Default 3

How many times are we trying to reconnect when communicating with Identity API Server.

cache

Type string

Default <None>

Request environment key where the Swift cache object is stored. When auth_token middleware is deployed with a Swift cache, use this option to have the middleware share a caching backend with swift. Otherwise, use the memcached_servers option instead.

certfile

Type string

Default <None>

Required if identity server requires client certificate

keyfile

Type string

Default <None>

Required if identity server requires client certificate

`cafile`

Type string

Default <None>

A PEM encoded Certificate Authority to use when verifying HTTPs connections. Defaults to system CAs.

`insecure`

Type boolean

Default False

Verify HTTPS connections.

`region_name`

Type string

Default <None>

The region in which the identity server can be found.

`memcached_servers`

Type list

Default <None>

Optionally specify a list of memcached server(s) to use for caching. If left undefined, tokens will instead be cached in-process.

Table 34: Deprecated Variations

Group	Name
keystone_authtoken	memcache_servers

`token_cache_time`

Type integer

Default 300

In order to prevent excessive effort spent validating tokens, the middleware caches previously-seen tokens for a configurable duration (in seconds). Set to -1 to disable caching completely.

`memcache_security_strategy`

Type string

Default None

Valid Values None, MAC, ENCRYPT

(Optional) If defined, indicate whether token data should be authenticated or authenticated and encrypted. If MAC, token data is authenticated (with HMAC) in the cache. If ENCRYPT, token data is encrypted and authenticated in the cache. If the value is not one of these options or empty, auth_token will raise an exception on initialization.

`memcache_secret_key`

Type string

Default `<None>`

(Optional, mandatory if memcache_security_strategy is defined) This string is used for key derivation.

`memcache_pool_dead_retry`

Type integer

Default `300`

(Optional) Number of seconds memcached server is considered dead before it is tried again.

`memcache_pool_maxsize`

Type integer

Default `10`

(Optional) Maximum total number of open connections to every memcached server.

`memcache_pool_socket_timeout`

Type integer

Default `3`

(Optional) Socket timeout in seconds for communicating with a memcached server.

`memcache_pool_unused_timeout`

Type integer

Default `60`

(Optional) Number of seconds a connection to memcached is held unused in the pool before it is closed.

`memcache_pool_conn_get_timeout`

Type integer

Default `10`

(Optional) Number of seconds that an operation will wait to get a memcached client connection from the pool.

memcache_use_advanced_pool

Type boolean

Default `True`

(Optional) Use the advanced (eventlet safe) memcached client pool.

include_service_catalog

Type boolean

Default `True`

(Optional) Indicate whether to set the X-Service-Catalog header. If False, middleware will not ask for service catalog on token validation and will not set the X-Service-Catalog header.

enforce_token_bind

Type string

Default `permissive`

Used to control the use and type of token binding. Can be set to: disabled to not check token binding. permissive (default) to validate binding information if the bind type is of a form known to the server and ignore it if not. strict like permissive but if the bind type is unknown the token will be rejected. required any form of token binding is needed to be allowed. Finally the name of a binding method that must be present in tokens.

service_token_roles

Type list

Default `['service']`

A choice of roles that must be present in a service token. Service tokens are allowed to request that an expired token can be used and so this check should tightly control that only actual services should be sending this token. Roles here are applied as an ANY check so any role in this list must be present. For backwards compatibility reasons this currently only affects the allow_expired check.

service_token_roles_required

Type boolean

Default `False`

For backwards compatibility reasons we must let valid service tokens pass that dont pass the service_token_roles check as valid. Setting this true will become the default in a future release and should be enabled if possible.

service_type

Type string

Default `<None>`

The name or type of the service as it appears in the service catalog. This is used to validate tokens that have restricted access rules.

auth_type

Type unknown type

Default <None>

Authentication type to load

Table 35: Deprecated Variations

Group	Name
keystone_authtoken	auth_plugin

auth_section

Type unknown type

Default <None>

Config Section from which to load plugin specific options

mdns

registration_attempts

Type integer

Default 5

Minimum Value 1

Number of attempts to register a service. Currently has to be larger than 1 because of race conditions in the zeroconf library.

lookup_attempts

Type integer

Default 3

Minimum Value 1

Number of attempts to lookup a service.

params

Type unknown type

Default {}

Additional parameters to pass for the registered service.

interfaces

Type list

Default <None>

List of IP addresses of interfaces to use for mDNS. Defaults to all interfaces on the system.

metrics

backend

Type string

Default noop

Valid Values noop, statsd

Backend to use for the metrics system.

prepend_host

Type boolean

Default False

Prepend the hostname to all metric names. The format of metric names is [global_prefix.][host_name.]prefix.metric_name.

prepend_host_reverse

Type boolean

Default True

Split the prepended host value by . and reverse it (to better match the reverse hierarchical form of domain names).

global_prefix

Type string

Default <None>

Prefix all metric names with this value. By default, there is no global prefix. The format of metric names is [global_prefix.][host_name.]prefix.metric_name.

agent_backend

Type string

Default noop

Backend for the agent ramdisk to use for metrics. Default possible backends are noop and statsd.

agent_prepend_host

Type boolean

Default `False`

Prepend the hostname to all metric names sent by the agent ramdisk. The format of metric names is [global_prefix.][uuid.][host_name.]prefix.metric_name.

`agent_prepend_uuid`

Type boolean

Default `False`

Prepend the nodes Ironic uuid to all metric names sent by the agent ramdisk. The format of metric names is [global_prefix.][uuid.][host_name.]prefix.metric_name.

`agent_prepend_host_reverse`

Type boolean

Default `True`

Split the prepended host value by . and reverse it for metrics sent by the agent ramdisk (to better match the reverse hierarchical form of domain names).

`agent_global_prefix`

Type string

Default `<None>`

Prefix all metric names sent by the agent ramdisk with this value. The format of metric names is [global_prefix.][uuid.][host_name.]prefix.metric_name.

metrics_statsd

`statsd_host`

Type string

Default `localhost`

Host for use with the statsd backend.

`statsd_port`

Type port number

Default `8125`

Minimum Value 0

Maximum Value 65535

Port to use with the statsd backend.

`agent_statsd_host`

Type string

Default `localhost`

Host for the agent ramdisk to use with the statsd backend. This must be accessible from networks the agent is booted on.

`agent_statsd_port`

Type port number

Default `8125`

Minimum Value 0

Maximum Value 65535

Port for the agent ramdisk to use with the statsd backend.

molds

`storage`

Type string

Default `swift`

Configuration mold storage location. Supports swift and http. By default swift.

`user`

Type string

Default `<None>`

User for http Basic auth. By default set empty.

`password`

Type string

Default `<None>`

Password for http Basic auth. By default set empty.

`retry_attempts`

Type integer

Default `3`

Retry attempts for saving or getting configuration molds.

`retry_interval`

Type integer

Default `3`

Retry interval for saving or getting configuration molds.

neutron

add_all_ports

Type boolean

Default `False`

Mutable This option can be changed without restarting.

Option to enable transmission of all ports to neutron when creating ports for provisioning, cleaning, or rescue. This is done without IP addresses assigned to the port, and may be useful in some bonded network configurations.

auth_url

Type unknown type

Default <None>

Authentication URL

auth_type

Type unknown type

Default <None>

Authentication type to load

Table 36: Deprecated Variations

Group	Name
neutron	auth_plugin

cafile

Type string

Default <None>

PEM encoded Certificate Authority to use when verifying HTTPs connections.

certfile

Type string

Default <None>

PEM encoded client certificate cert file

cleaning_network

Type string

Default <None>

Mutable This option can be changed without restarting.

Neutron network UUID or name for the ramdisk to be booted into for cleaning nodes. Required for neutron network interface. It is also required if cleaning nodes when using flat network interface or neutron DHCP provider. If a name is provided, it must be unique among all networks or cleaning will fail.

Table 37: Deprecated Variations

Group	Name
neutron	cleaning_network_uuid

`cleaning_network_security_groups`

Type list

Default []

Mutable This option can be changed without restarting.

List of Neutron Security Group UUIDs to be applied during cleaning of the nodes. Optional for the neutron network interface and not used for the flat or noop network interfaces. If not specified, default security group is used.

`collect_timing`

Type boolean

Default False

Collect per-API call timing information.

`connect_retries`

Type integer

Default <None>

The maximum number of retries that should be attempted for connection errors.

`connect_retry_delay`

Type floating point

Default <None>

Delay (in seconds) between two retries for connection errors. If not set, exponential retry starting with 0.5 seconds up to a maximum of 60 seconds is used.

`default_domain_id`

Type unknown type

Default <None>

Optional domain ID to use with v3 and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

default_domain_name

Type unknown type

Default <None>

Optional domain name to use with v3 API and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

dhcpv6_stateful_address_count

Type integer

Default 4

Mutable This option can be changed without restarting.

Number of IPv6 addresses to allocate for ports created for provisioning, cleaning, rescue or inspection on DHCPv6-stateful networks. Different stages of the chain-loading process will request addresses with different CLID/IAID. Due to non-identical identifiers multiple addresses must be reserved for the host to ensure each step of the boot process can successfully lease addresses.

domain_id

Type unknown type

Default <None>

Domain ID to scope to

domain_name

Type unknown type

Default <None>

Domain name to scope to

endpoint_override

Type string

Default <None>

Always use this endpoint URL for requests for this client. NOTE: The unversioned endpoint should be specified here; to request a particular API version, use the *version*, *min-version*, and/or *max-version* options.

insecure

Type boolean

Default False

Verify HTTPS connections.

`inspection_network`

Type string

Default <None>

Mutable This option can be changed without restarting.

Neutron network UUID or name for the ramdisk to be booted into for in-band inspection of nodes. If a name is provided, it must be unique among all networks or inspection will fail.

`inspection_network_security_groups`

Type list

Default []

Mutable This option can be changed without restarting.

List of Neutron Security Group UUIDs to be applied during the node inspection process. Optional for the neutron network interface and not used for the flat or noop network interfaces. If not specified, the default security group is used.

`keyfile`

Type string

Default <None>

PEM encoded client certificate key file

`max_version`

Type string

Default <None>

The maximum major version of a given API, intended to be used as the upper bound of a range with min_version. Mutually exclusive with version.

`min_version`

Type string

Default <None>

The minimum major version of a given API, intended to be used as the lower bound of a range with max_version. Mutually exclusive with version. If min_version is given with no max_version it is as if max version is latest.

`password`

Type unknown type

Default <None>

Users password

`port_setup_delay`

Type integer

Default `0`

Minimum Value 0

Mutable This option can be changed without restarting.

Delay value to wait for Neutron agents to setup sufficient DHCP configuration for port.

`project_domain_id`

Type unknown type

Default `<None>`

Domain ID containing project

`project_domain_name`

Type unknown type

Default `<None>`

Domain name containing project

`project_id`

Type unknown type

Default `<None>`

Project ID to scope to

Table 38: Deprecated Variations

Group	Name
neutron	tenant-id
neutron	tenant_id

`project_name`

Type unknown type

Default `<None>`

Project name to scope to

Table 39: Deprecated Variations

Group	Name
neutron	tenant-name
neutron	tenant_name

`provisioning_network`

Type string

Default <None>

Mutable This option can be changed without restarting.

Neutron network UUID or name for the ramdisk to be booted into for provisioning nodes. Required for neutron network interface. If a name is provided, it must be unique among all networks or deploy will fail.

Table 40: Deprecated Variations

Group	Name
neutron	provisioning_network_uuid

`provisioning_network_security_groups`

Type list

Default []

Mutable This option can be changed without restarting.

List of Neutron Security Group UUIDs to be applied during provisioning of the nodes. Optional for the neutron network interface and not used for the flat or noop network interfaces. If not specified, default security group is used.

`region_name`

Type string

Default <None>

The default region_name for endpoint URL discovery.

`request_timeout`

Type integer

Default 45

Mutable This option can be changed without restarting.

Timeout for request processing when interacting with Neutron. This value should be increased if neutron port action timeouts are observed as neutron performs pre-commit validation prior returning to the API client which can take longer than normal client/server interactions.

`rescuing_network`

Type string

Default <None>

Mutable This option can be changed without restarting.

Neutron network UUID or name for booting the ramdisk for rescue mode. This is not the network that the rescue ramdisk will use post-boot the tenant network is used for that. Required for neutron

network interface, if rescue mode will be used. It is not used for the flat or noop network interfaces. If a name is provided, it must be unique among all networks or rescue will fail.

`rescuing_network_security_groups`

Type list

Default `[]`

Mutable This option can be changed without restarting.

List of Neutron Security Group UUIDs to be applied during the node rescue process. Optional for the neutron network interface and not used for the flat or noop network interfaces. If not specified, the default security group is used.

`retries`

Type integer

Default `3`

Mutable This option can be changed without restarting.

DEPRECATED: Client retries in the case of a failed request.

> **Warning:** This option is deprecated for removal. Its value may be silently ignored in the future.
>
> **Reason** Replaced by status_code_retries and status_code_retry_delay.

`service_name`

Type string

Default `<None>`

The default service_name for endpoint URL discovery.

`service_type`

Type string

Default `network`

The default service_type for endpoint URL discovery.

`split_loggers`

Type boolean

Default `False`

Log requests to multiple loggers.

`status_code_retries`

Type integer

Default <None>

The maximum number of retries that should be attempted for retriable HTTP status codes.

`status_code_retry_delay`

Type floating point

Default <None>

Delay (in seconds) between two retries for retriable status codes. If not set, exponential retry starting with 0.5 seconds up to a maximum of 60 seconds is used.

`system_scope`

Type unknown type

Default <None>

Scope for system operations

`tenant_id`

Type unknown type

Default <None>

Tenant ID

`tenant_name`

Type unknown type

Default <None>

Tenant Name

`timeout`

Type integer

Default <None>

Timeout value for http requests

`trust_id`

Type unknown type

Default <None>

Trust ID

`user_domain_id`

Type unknown type

Default <None>

Users domain id

user_domain_name

Type unknown type

Default <None>

Users domain name

user_id

Type unknown type

Default <None>

User id

username

Type unknown type

Default <None>

Username

Table 41: Deprecated Variations

Group	Name
neutron	user-name
neutron	user_name

valid_interfaces

Type list

Default ['internal', 'public']

List of interfaces, in order of preference, for endpoint URL.

version

Type string

Default <None>

Minimum Major API version within a given Major API version for endpoint URL discovery. Mutually exclusive with min_version and max_version

nova

auth_url

Type unknown type

Default <None>

Authentication URL

auth_type

Type unknown type

Default <None>

Authentication type to load

Table 42: Deprecated Variations

Group	Name
nova	auth_plugin

cafile

Type string

Default <None>

PEM encoded Certificate Authority to use when verifying HTTPs connections.

certfile

Type string

Default <None>

PEM encoded client certificate cert file

collect_timing

Type boolean

Default False

Collect per-API call timing information.

connect_retries

Type integer

Default <None>

The maximum number of retries that should be attempted for connection errors.

connect_retry_delay

Type floating point

Default <None>

Delay (in seconds) between two retries for connection errors. If not set, exponential retry starting with 0.5 seconds up to a maximum of 60 seconds is used.

default_domain_id

Type unknown type

Default <None>

Optional domain ID to use with v3 and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

default_domain_name

Type unknown type

Default <None>

Optional domain name to use with v3 API and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

domain_id

Type unknown type

Default <None>

Domain ID to scope to

domain_name

Type unknown type

Default <None>

Domain name to scope to

endpoint_override

Type string

Default <None>

Always use this endpoint URL for requests for this client. NOTE: The unversioned endpoint should be specified here; to request a particular API version, use the *version*, *min-version*, and/or *max-version* options.

insecure

Type boolean

Default False

Verify HTTPS connections.

keyfile

Type string

Default <None>

PEM encoded client certificate key file

max_version

Type string

Default <None>

The maximum major version of a given API, intended to be used as the upper bound of a range with min_version. Mutually exclusive with version.

min_version

Type string

Default <None>

The minimum major version of a given API, intended to be used as the lower bound of a range with max_version. Mutually exclusive with version. If min_version is given with no max_version it is as if max version is latest.

password

Type unknown type

Default <None>

Users password

project_domain_id

Type unknown type

Default <None>

Domain ID containing project

project_domain_name

Type unknown type

Default <None>

Domain name containing project

project_id

Type unknown type

Default <None>

Project ID to scope to

Table 43: Deprecated Variations

Group	Name
nova	tenant-id
nova	tenant_id

project_name

Type unknown type

Default <None>

Project name to scope to

Table 44: Deprecated Variations

Group	Name
nova	tenant-name
nova	tenant_name

region_name

Type string

Default <None>

The default region_name for endpoint URL discovery.

send_power_notifications

Type boolean

Default True

Mutable This option can be changed without restarting.

When set to True, it will enable the support for power state change callbacks to nova. This option should be set to False in deployments that do not have the openstack compute service.

service_name

Type string

Default <None>

The default service_name for endpoint URL discovery.

service_type

Type string

Default compute

The default service_type for endpoint URL discovery.

`split_loggers`

Type boolean

Default `False`

Log requests to multiple loggers.

`status_code_retries`

Type integer

Default <None>

The maximum number of retries that should be attempted for retriable HTTP status codes.

`status_code_retry_delay`

Type floating point

Default <None>

Delay (in seconds) between two retries for retriable status codes. If not set, exponential retry starting with 0.5 seconds up to a maximum of 60 seconds is used.

`system_scope`

Type unknown type

Default <None>

Scope for system operations

`tenant_id`

Type unknown type

Default <None>

Tenant ID

`tenant_name`

Type unknown type

Default <None>

Tenant Name

`timeout`

Type integer

Default <None>

Timeout value for http requests

`trust_id`

Type unknown type

Default <None>

Trust ID

user_domain_id

Type unknown type

Default <None>

Users domain id

user_domain_name

Type unknown type

Default <None>

Users domain name

user_id

Type unknown type

Default <None>

User id

username

Type unknown type

Default <None>

Username

Table 45: Deprecated Variations

Group	Name
nova	user-name
nova	user_name

valid_interfaces

Type list

Default ['internal', 'public']

List of interfaces, in order of preference, for endpoint URL.

version

Type string

Default <None>

Minimum Major API version within a given Major API version for endpoint URL discovery. Mutually exclusive with min_version and max_version

oslo_concurrency

disable_process_locking

Type boolean

Default `False`

Enables or disables inter-process locks.

Table 46: Deprecated Variations

Group	Name
DEFAULT	disable_process_locking

lock_path

Type string

Default `<None>`

Directory to use for lock files. For security, the specified directory should only be writable by the user running the processes that need locking. Defaults to environment variable OSLO_LOCK_PATH. If external locks are used, a lock path must be set.

Table 47: Deprecated Variations

Group	Name
DEFAULT	lock_path

oslo_messaging_amqp

container_name

Type string

Default `<None>`

Name for the AMQP container. must be globally unique. Defaults to a generated UUID

Table 48: Deprecated Variations

Group	Name
amqp1	container_name

idle_timeout

Type integer

Default `0`

Timeout for inactive connections (in seconds)

Table 49: Deprecated Variations

Group	Name
amqp1	idle_timeout

trace

Type boolean

Default `False`

Debug: dump AMQP frames to stdout

Table 50: Deprecated Variations

Group	Name
amqp1	trace

ssl

Type boolean

Default `False`

Attempt to connect via SSL. If no other ssl-related parameters are given, it will use the systems CA-bundle to verify the servers certificate.

ssl_ca_file

Type string

Default `''`

CA certificate PEM file used to verify the servers certificate

Table 51: Deprecated Variations

Group	Name
amqp1	ssl_ca_file

ssl_cert_file

Type string

Default `''`

Self-identifying certificate PEM file for client authentication

Table 52: Deprecated Variations

Group	Name
amqp1	ssl_cert_file

ssl_key_file

Type string

Default ''

Private key PEM file used to sign ssl_cert_file certificate (optional)

Table 53: Deprecated Variations

Group	Name
amqp1	ssl_key_file

ssl_key_password

Type string

Default <None>

Password for decrypting ssl_key_file (if encrypted)

Table 54: Deprecated Variations

Group	Name
amqp1	ssl_key_password

ssl_verify_vhost

Type boolean

Default False

By default SSL checks that the name in the servers certificate matches the hostname in the transport_url. In some configurations it may be preferable to use the virtual hostname instead, for example if the server uses the Server Name Indication TLS extension (rfc6066) to provide a certificate per virtual host. Set ssl_verify_vhost to True if the servers SSL certificate uses the virtual host name instead of the DNS name.

sasl_mechanisms

Type string

Default ''

Space separated list of acceptable SASL mechanisms

Table 55: Deprecated Variations

Group	Name
amqp1	sasl_mechanisms

sasl_config_dir

Type string

Default ''

Path to directory that contains the SASL configuration

Table 56: Deprecated Variations

Group	Name
amqp1	sasl_config_dir

`sasl_config_name`

Type string

Default ''

Name of configuration file (without .conf suffix)

Table 57: Deprecated Variations

Group	Name
amqp1	sasl_config_name

`sasl_default_realm`

Type string

Default ''

SASL realm to use if no realm present in username

`connection_retry_interval`

Type integer

Default 1

Minimum Value 1

Seconds to pause before attempting to re-connect.

`connection_retry_backoff`

Type integer

Default 2

Minimum Value 0

Increase the connection_retry_interval by this many seconds after each unsuccessful failover attempt.

`connection_retry_interval_max`

Type integer

Default 30

Minimum Value 1

Maximum limit for connection_retry_interval + connection_retry_backoff

`link_retry_delay`

Type integer

Default `10`

Minimum Value 1

Time to pause between re-connecting an AMQP 1.0 link that failed due to a recoverable error.

`default_reply_retry`

Type integer

Default `0`

Minimum Value -1

The maximum number of attempts to re-send a reply message which failed due to a recoverable error.

`default_reply_timeout`

Type integer

Default `30`

Minimum Value 5

The deadline for an rpc reply message delivery.

`default_send_timeout`

Type integer

Default `30`

Minimum Value 5

The deadline for an rpc cast or call message delivery. Only used when caller does not provide a timeout expiry.

`default_notify_timeout`

Type integer

Default `30`

Minimum Value 5

The deadline for a sent notification message delivery. Only used when caller does not provide a timeout expiry.

`default_sender_link_timeout`

Type integer

Default `600`

Minimum Value 1

The duration to schedule a purge of idle sender links. Detach link after expiry.

`addressing_mode`

Type string

Default `dynamic`

Indicates the addressing mode used by the driver. Permitted values: legacy - use legacy non-routable addressing routable - use routable addresses dynamic - use legacy addresses if the message bus does not support routing otherwise use routable addressing

`pseudo_vhost`

Type boolean

Default `True`

Enable virtual host support for those message buses that do not natively support virtual hosting (such as qpidd). When set to true the virtual host name will be added to all message bus addresses, effectively creating a private subnet per virtual host. Set to False if the message bus supports virtual hosting using the hostname field in the AMQP 1.0 Open performative as the name of the virtual host.

`server_request_prefix`

Type string

Default `exclusive`

address prefix used when sending to a specific server

Table 58: Deprecated Variations

Group	Name
amqp1	server_request_prefix

`broadcast_prefix`

Type string

Default `broadcast`

address prefix used when broadcasting to all servers

Table 59: Deprecated Variations

Group	Name
amqp1	broadcast_prefix

group_request_prefix

Type string

Default `unicast`

address prefix when sending to any server in group

Table 60: Deprecated Variations

Group	Name
amqp1	group_request_prefix

rpc_address_prefix

Type string

Default `openstack.org/om/rpc`

Address prefix for all generated RPC addresses

notify_address_prefix

Type string

Default `openstack.org/om/notify`

Address prefix for all generated Notification addresses

multicast_address

Type string

Default `multicast`

Appended to the address prefix when sending a fanout message. Used by the message bus to identify fanout messages.

unicast_address

Type string

Default `unicast`

Appended to the address prefix when sending to a particular RPC/Notification server. Used by the message bus to identify messages sent to a single destination.

anycast_address

Type string

Default `anycast`

Appended to the address prefix when sending to a group of consumers. Used by the message bus to identify messages that should be delivered in a round-robin fashion across consumers.

default_notification_exchange

Type string

Default <None>

Exchange name used in notification addresses. Exchange name resolution precedence: Target.exchange if set else default_notification_exchange if set else control_exchange if set else notify

default_rpc_exchange

Type string

Default <None>

Exchange name used in RPC addresses. Exchange name resolution precedence: Target.exchange if set else default_rpc_exchange if set else control_exchange if set else rpc

reply_link_credit

Type integer

Default 200

Minimum Value 1

Window size for incoming RPC Reply messages.

rpc_server_credit

Type integer

Default 100

Minimum Value 1

Window size for incoming RPC Request messages

notify_server_credit

Type integer

Default 100

Minimum Value 1

Window size for incoming Notification messages

pre_settled

Type multi-valued

Default rpc-cast

Default rpc-reply

Send messages of this type pre-settled. Pre-settled messages will not receive acknowledgement from the peer. Note well: pre-settled messages may be silently discarded if the delivery fails. Permitted values: rpc-call - send RPC Calls pre-settled rpc-reply- send RPC Replies pre-settled rpc-cast - Send RPC Casts pre-settled notify - Send Notifications pre-settled

oslo_messaging_kafka

kafka_max_fetch_bytes

Type integer

Default `1048576`

Max fetch bytes of Kafka consumer

kafka_consumer_timeout

Type floating point

Default `1.0`

Default timeout(s) for Kafka consumers

pool_size

Type integer

Default `10`

Pool Size for Kafka Consumers

> **Warning:** This option is deprecated for removal. Its value may be silently ignored in the future.
>
> **Reason** Driver no longer uses connection pool.

conn_pool_min_size

Type integer

Default `2`

The pool size limit for connections expiration policy

> **Warning:** This option is deprecated for removal. Its value may be silently ignored in the future.
>
> **Reason** Driver no longer uses connection pool.

conn_pool_ttl

Type integer

Default `1200`

The time-to-live in sec of idle connections in the pool

> **Warning:** This option is deprecated for removal. Its value may be silently ignored in the future.
>
> **Reason** Driver no longer uses connection pool.

consumer_group

Type string

Default `oslo_messaging_consumer`

Group id for Kafka consumer. Consumers in one group will coordinate message consumption

producer_batch_timeout

Type floating point

Default `0.0`

Upper bound on the delay for KafkaProducer batching in seconds

producer_batch_size

Type integer

Default `16384`

Size of batch for the producer async send

compression_codec

Type string

Default `none`

Valid Values none, gzip, snappy, lz4, zstd

The compression codec for all data generated by the producer. If not set, compression will not be used. Note that the allowed values of this depend on the kafka version

enable_auto_commit

Type boolean

Default `False`

Enable asynchronous consumer commits

max_poll_records

Type integer

Default `500`

The maximum number of records returned in a poll call

security_protocol

Type string

Default `PLAINTEXT`

Valid Values PLAINTEXT, SASL_PLAINTEXT, SSL, SASL_SSL

Protocol used to communicate with brokers

sasl_mechanism

Type string

Default `PLAIN`

Mechanism when security protocol is SASL

ssl_cafile

Type string

Default `''`

CA certificate PEM file used to verify the server certificate

ssl_client_cert_file

Type string

Default `''`

Client certificate PEM file used for authentication.

ssl_client_key_file

Type string

Default `''`

Client key PEM file used for authentication.

ssl_client_key_password

Type string

Default `''`

Client key password file used for authentication.

oslo_messaging_notifications

driver

Type multi-valued

Default ''

The Drivers(s) to handle sending notifications. Possible values are messaging, messagingv2, routing, log, test, noop

Table 61: Deprecated Variations

Group	Name
DEFAULT	notification_driver

transport_url

Type string

Default <None>

A URL representing the messaging driver to use for notifications. If not set, we fall back to the same configuration used for RPC.

Table 62: Deprecated Variations

Group	Name
DEFAULT	notification_transport_url

topics

Type list

Default ['notifications']

AMQP topic used for OpenStack notifications.

Table 63: Deprecated Variations

Group	Name
rpc_notifier2	topics
DEFAULT	notification_topics

retry

Type integer

Default -1

The maximum number of attempts to re-send a notification message which failed to be delivered due to a recoverable error. 0 - No retry, -1 - indefinite

oslo_messaging_rabbit

amqp_durable_queues

Type boolean

Default `False`

Use durable queues in AMQP. If rabbit_quorum_queue is enabled, queues will be durable and this value will be ignored.

amqp_auto_delete

Type boolean

Default `False`

Auto-delete queues in AMQP.

Table 64: Deprecated Variations

Group	Name
DEFAULT	amqp_auto_delete

ssl

Type boolean

Default `False`

Connect over SSL.

Table 65: Deprecated Variations

Group	Name
oslo_messaging_rabbit	rabbit_use_ssl

ssl_version

Type string

Default `''`

SSL version to use (valid only if SSL enabled). Valid values are TLSv1 and SSLv23. SSLv2, SSLv3, TLSv1_1, and TLSv1_2 may be available on some distributions.

Table 66: Deprecated Variations

Group	Name
oslo_messaging_rabbit	kombu_ssl_version

ssl_key_file

Type string

Default ''

SSL key file (valid only if SSL enabled).

Table 67: Deprecated Variations

Group	Name
oslo_messaging_rabbit	kombu_ssl_keyfile

`ssl_cert_file`

Type string

Default ''

SSL cert file (valid only if SSL enabled).

Table 68: Deprecated Variations

Group	Name
oslo_messaging_rabbit	kombu_ssl_certfile

`ssl_ca_file`

Type string

Default ''

SSL certification authority file (valid only if SSL enabled).

Table 69: Deprecated Variations

Group	Name
oslo_messaging_rabbit	kombu_ssl_ca_certs

`heartbeat_in_pthread`

Type boolean

Default True

Run the health check heartbeat thread through a native python thread by default. If this option is equal to False then the health check heartbeat will inherit the execution model from the parent process. For example if the parent process has monkey patched the stdlib by using eventlet/greenlet then the heartbeat will be run through a green thread.

`kombu_reconnect_delay`

Type floating point

Default 1.0

Minimum Value 0.0

Maximum Value 4.5

How long to wait (in seconds) before reconnecting in response to an AMQP consumer cancel notification.

Table 70: Deprecated Variations

Group	Name
DEFAULT	kombu_reconnect_delay

kombu_compression

Type string

Default <None>

EXPERIMENTAL: Possible values are: gzip, bz2. If not set compression will not be used. This option may not be available in future versions.

kombu_missing_consumer_retry_timeout

Type integer

Default 60

How long to wait a missing client before abandoning to send it its replies. This value should not be longer than rpc_response_timeout.

Table 71: Deprecated Variations

Group	Name
oslo_messaging_rabbit	kombu_reconnect_timeout

kombu_failover_strategy

Type string

Default round-robin

Valid Values round-robin, shuffle

Determines how the next RabbitMQ node is chosen in case the one we are currently connected to becomes unavailable. Takes effect only if more than one RabbitMQ node is provided in config.

rabbit_login_method

Type string

Default AMQPLAIN

Valid Values PLAIN, AMQPLAIN, RABBIT-CR-DEMO

The RabbitMQ login method.

Table 72: Deprecated Variations

Group	Name
DEFAULT	rabbit_login_method

rabbit_retry_interval

Type integer

Default 1

How frequently to retry connecting with RabbitMQ.

rabbit_retry_backoff

Type integer

Default 2

How long to backoff for between retries when connecting to RabbitMQ.

Table 73: Deprecated Variations

Group	Name
DEFAULT	rabbit_retry_backoff

rabbit_interval_max

Type integer

Default 30

Maximum interval of RabbitMQ connection retries. Default is 30 seconds.

rabbit_ha_queues

Type boolean

Default False

Try to use HA queues in RabbitMQ (x-ha-policy: all). If you change this option, you must wipe the RabbitMQ database. In RabbitMQ 3.0, queue mirroring is no longer controlled by the x-ha-policy argument when declaring a queue. If you just want to make sure that all queues (except those with auto-generated names) are mirrored across all nodes, run: rabbitmqctl set_policy HA ^(?!amq.).* {ha-mode: all}

Table 74: Deprecated Variations

Group	Name
DEFAULT	rabbit_ha_queues

rabbit_quorum_queue

Type boolean

Default False

Use quorum queues in RabbitMQ (x-queue-type: quorum). The quorum queue is a modern queue type for RabbitMQ implementing a durable, replicated FIFO queue based on the Raft consensus algorithm. It is available as of RabbitMQ 3.8.0. If set this option will conflict with the HA queues

(rabbit_ha_queues) aka mirrored queues, in other words the HA queues should be disabled, quorum queues durable by default so the amqp_durable_queues opion is ignored when this option enabled.

rabbit_transient_queues_ttl

Type integer

Default 1800

Minimum Value 1

Positive integer representing duration in seconds for queue TTL (x-expires). Queues which are unused for the duration of the TTL are automatically deleted. The parameter affects only reply and fanout queues.

rabbit_qos_prefetch_count

Type integer

Default 0

Specifies the number of messages to prefetch. Setting to zero allows unlimited messages.

heartbeat_timeout_threshold

Type integer

Default 60

Number of seconds after which the Rabbit broker is considered down if heartbeats keep-alive fails (0 disables heartbeat).

heartbeat_rate

Type integer

Default 2

How often times during the heartbeat_timeout_threshold we check the heartbeat.

direct_mandatory_flag

Type boolean

Default True

(DEPRECATED) Enable/Disable the RabbitMQ mandatory flag for direct send. The direct send is used as reply, so the MessageUndeliverable exception is raised in case the client queue does not exist.MessageUndeliverable exception will be used to loop for a timeout to lets a chance to sender to recover.This flag is deprecated and it will not be possible to deactivate this functionality anymore

Warning: This option is deprecated for removal. Its value may be silently ignored in the future.

Reason Mandatory flag no longer deactivable.

enable_cancel_on_failover

Type boolean

Default `False`

Enable x-cancel-on-ha-failover flag so that rabbitmq server will cancel and notify consumerswhen queue is down

oslo_middleware

enable_proxy_headers_parsing

Type boolean

Default `False`

Whether the application is behind a proxy or not. This determines if the middleware should parse the headers or not.

oslo_policy

enforce_scope

Type boolean

Default `False`

This option controls whether or not to enforce scope when evaluating policies. If `True`, the scope of the token used in the request is compared to the `scope_types` of the policy being enforced. If the scopes do not match, an `InvalidScope` exception will be raised. If `False`, a message will be logged informing operators that policies are being invoked with mismatching scope.

enforce_new_defaults

Type boolean

Default `False`

This option controls whether or not to use old deprecated defaults when evaluating policies. If `True`, the old deprecated defaults are not going to be evaluated. This means if any existing token is allowed for old defaults but is disallowed for new defaults, it will be disallowed. It is encouraged to enable this flag along with the `enforce_scope` flag so that you can get the benefits of new defaults and `scope_type` together. If `False`, the deprecated policy check string is logically ORd with the new policy check string, allowing for a graceful upgrade experience between releases with new policies, which is the default behavior.

policy_file

Type string

Default `policy.yaml`

The relative or absolute path of a file that maps roles to permissions for a given service. Relative paths must be specified in relation to the configuration file setting this option.

Table 75: Deprecated Variations

Group	Name
DEFAULT	policy_file

policy_default_rule

Type string

Default `default`

Default rule. Enforced when a requested rule is not found.

Table 76: Deprecated Variations

Group	Name
DEFAULT	policy_default_rule

policy_dirs

Type multi-valued

Default `policy.d`

Directories where policy configuration files are stored. They can be relative to any directory in the search path defined by the config_dir option, or absolute paths. The file defined by policy_file must exist for these directories to be searched. Missing or empty directories are ignored.

Table 77: Deprecated Variations

Group	Name
DEFAULT	policy_dirs

remote_content_type

Type string

Default `application/x-www-form-urlencoded`

Valid Values application/x-www-form-urlencoded, application/json

Content Type to send and receive data for REST based policy check

remote_ssl_verify_server_crt

Type boolean

Default `False`

server identity verification for REST based policy check

remote_ssl_ca_crt_file

Type string

Default <None>

Absolute path to ca cert file for REST based policy check

remote_ssl_client_crt_file

Type string

Default <None>

Absolute path to client cert for REST based policy check

remote_ssl_client_key_file

Type string

Default <None>

Absolute path client key file REST based policy check

profiler

enabled

Type boolean

Default False

Enable the profiling for all services on this node.

Default value is False (fully disable the profiling feature).

Possible values:

- True: Enables the feature
- False: Disables the feature. The profiling cannot be started via this project operations. If the profiling is triggered by another project, this project part will be empty.

Table 78: Deprecated Variations

Group	Name
profiler	profiler_enabled

trace_sqlalchemy

Type boolean

Default False

Enable SQL requests profiling in services.

Default value is False (SQL requests wont be traced).

Possible values:

- True: Enables SQL requests profiling. Each SQL query will be part of the trace and can the be analyzed by how much time was spent for that.
- False: Disables SQL requests profiling. The spent time is only shown on a higher level of operations. Single SQL queries cannot be analyzed this way.

hmac_keys

Type string

Default `SECRET_KEY`

Secret key(s) to use for encrypting context data for performance profiling.

This string value should have the following format: <key1>[,<key2>,<keyn>], where each key is some random string. A user who triggers the profiling via the REST API has to set one of these keys in the headers of the REST API call to include profiling results of this node for this particular project.

Both enabled flag and hmac_keys config options should be set to enable profiling. Also, to generate correct profiling information across all services at least one key needs to be consistent between OpenStack projects. This ensures it can be used from client side to generate the trace, containing information from all possible resources.

connection_string

Type string

Default `messaging://`

Connection string for a notifier backend.

Default value is `messaging://` which sets the notifier to oslo_messaging.

Examples of possible values:

- `messaging://` - use oslo_messaging driver for sending spans.
- `redis://127.0.0.1:6379` - use redis driver for sending spans.
- `mongodb://127.0.0.1:27017` - use mongodb driver for sending spans.
- `elasticsearch://127.0.0.1:9200` - use elasticsearch driver for sending spans.
- `jaeger://127.0.0.1:6831` - use jaeger tracing as driver for sending spans.

es_doc_type

Type string

Default `notification`

Document type for notification indexing in elasticsearch.

es_scroll_time

Type string

Default `2m`

This parameter is a time value parameter (for example: es_scroll_time=2m), indicating for how long the nodes that participate in the search will maintain relevant resources in order to continue and support it.

es_scroll_size

Type integer

Default `10000`

Elasticsearch splits large requests in batches. This parameter defines maximum size of each batch (for example: es_scroll_size=10000).

socket_timeout

Type floating point

Default `0.1`

Redissentinel provides a timeout option on the connections. This parameter defines that timeout (for example: socket_timeout=0.1).

sentinel_service_name

Type string

Default `mymaster`

Redissentinel uses a service name to identify a master redis service. This parameter defines the name (for example: `sentinal_service_name=mymaster`).

filter_error_trace

Type boolean

Default `False`

Enable filter traces that contain error/exception to a separated place.

Default value is set to False.

Possible values:

- True: Enable filter traces that contain error/exception.
- False: Disable the filter.

pxe

kernel_append_params

Type string

Default `nofb nomodeset vga=normal`

Mutable This option can be changed without restarting.

Additional append parameters for baremetal PXE boot.

Table 79: Deprecated Variations

Group	Name
pxe	pxe_append_params

default_ephemeral_format

Type string

Default `ext4`

Mutable This option can be changed without restarting.

Default file system format for ephemeral partition, if one is created.

images_path

Type string

Default `/var/lib/ironic/images/`

On the ironic-conductor node, directory where images are stored on disk.

instance_master_path

Type string

Default `/var/lib/ironic/master_images`

On the ironic-conductor node, directory where master instance images are stored on disk. Setting to the empty string disables image caching.

image_cache_size

Type integer

Default `20480`

Maximum size (in MiB) of cache for master images, including those in use.

image_cache_ttl

Type integer

Default `10080`

Maximum TTL (in minutes) for old master images in cache.

pxe_config_template

Type string

Default `$pybasedir/drivers/modules/pxe_config.template`

Mutable This option can be changed without restarting.

On ironic-conductor node, template file for PXE loader configuration.

`ipxe_config_template`

Type string

Default `$pybasedir/drivers/modules/ipxe_config.template`

Mutable This option can be changed without restarting.

On ironic-conductor node, template file for iPXE operations.

`uefi_pxe_config_template`

Type string

Default `$pybasedir/drivers/modules/pxe_grub_config.template`

Mutable This option can be changed without restarting.

On ironic-conductor node, template file for PXE configuration for UEFI boot loader. Generally this is used for GRUB specific templates.

`pxe_config_template_by_arch`

Type dict

Default `{}`

Mutable This option can be changed without restarting.

On ironic-conductor node, template file for PXE configuration per node architecture. For example: aarch64:/opt/share/grubaa64_pxe_config.template

`tftp_server`

Type string

Default `$my_ip`

IP address of ironic-conductor nodes TFTP server.

`tftp_root`

Type string

Default `/tftpboot`

ironic-conductor nodes TFTP root path. The ironic-conductor must have read/write access to this path.

`tftp_master_path`

Type string

Default `/tftpboot/master_images`

On ironic-conductor node, directory where master TFTP images are stored on disk. Setting to the empty string disables image caching.

dir_permission

Type integer

Default <None>

The permission that will be applied to the TFTP folders upon creation. This should be set to the permission such that the tftpserver has access to read the contents of the configured TFTP folder. This setting is only required when the operating systems umask is restrictive such that ironic-conductor is creating files that cannot be read by the TFTP server. Setting to <None> will result in the operating systems umask to be utilized for the creation of new tftp folders. The system default umask is masked out on the specified value. It is required that an octal representation is specified. For example: 0o755

file_permission

Type integer

Default 420

The permission which is used on files created as part of configuration and setup of file assets for PXE based operations. Defaults to a value of 0o644. This value must be specified as an octal representation. For example: 0o644

pxe_bootfile_name

Type string

Default pxelinux.0

Bootfile DHCP parameter.

pxe_config_subdir

Type string

Default pxelinux.cfg

Directory in which to create symbolic links which represent the MAC or IP address of the ports on a node and allow boot loaders to load the PXE file for the node. This directory name is relative to the PXE or iPXE folders.

uefi_pxe_bootfile_name

Type string

Default bootx64.efi

Bootfile DHCP parameter for UEFI boot mode.

ipxe_bootfile_name

Type string

Default undionly.kpxe

Bootfile DHCP parameter.

`uefi_ipxe_bootfile_name`

Type string

Default `snponly.efi`

Bootfile DHCP parameter for UEFI boot mode. If you experience problems with booting using it, try ipxe.efi.

`pxe_bootfile_name_by_arch`

Type dict

Default `{}`

Bootfile DHCP parameter per node architecture. For example: aarch64:grubaa64.efi

`ipxe_bootfile_name_by_arch`

Type dict

Default `{}`

Bootfile DHCP parameter per node architecture. For example: aarch64:ipxe_aa64.efi

`ipxe_boot_script`

Type string

Default `$pybasedir/drivers/modules/boot.ipxe`

On ironic-conductor node, the path to the main iPXE script file.

`ipxe_fallback_script`

Type string

Default `<None>`

File name (e.g. inspector.ipxe) of an iPXE script to fall back to when booting to a MAC-specific script fails. When not set, booting will fail in this case.

`ipxe_timeout`

Type integer

Default `0`

Timeout value (in seconds) for downloading an image via iPXE. Defaults to 0 (no timeout)

`boot_retry_timeout`

Type integer

Default `<None>`

Minimum Value 60

Timeout (in seconds) after which PXE boot should be retried. Must be less than [conductor]deploy_callback_timeout. Disabled by default.

`boot_retry_check_interval`

Type integer

Default `90`

Minimum Value 1

Interval (in seconds) between periodic checks on PXE boot retry. Has no effect if boot_retry_timeout is not set.

`ip_version`

Type string

Default `4`

Valid Values 4, 6

Mutable This option can be changed without restarting.

The IP version that will be used for PXE booting. Defaults to 4. This option has been a no-op for in-treedrivers since the Ussuri development cycle.

Possible values

4 IPv4

6 IPv6

> **Warning:** This option is deprecated for removal. Its value may be silently ignored in the future.

`ipxe_use_swift`

Type boolean

Default `False`

Mutable This option can be changed without restarting.

Download deploy and rescue images directly from swift using temporary URLs. If set to false (default), images are downloaded to the ironic-conductor node and served over its local HTTP server. Applicable only when ipxe compatible boot interface is used.

`enable_netboot_fallback`

Type boolean

Default `False`

Mutable This option can be changed without restarting.

If True, generate a PXE environment even for nodes that use local boot. This is useful when the driver cannot switch nodes to local boot, e.g. with SNMP or with Redfish on machines that cannot do persistent boot. Mostly useful for standalone ironic since Neutron will prevent incorrect PXE boot.

`loader_file_paths`

Type unknown type

Default `{}`

Dictionary describing the bootloaders to load into conductor PXE/iPXE boot folders values from the host operating system. Formatted as key of destination file name, and value of a full path to a file to be copied. File assets will have [pxe]file_permission applied, if set. If used, the file names should match established bootloader configuration settings for bootloaders. Use example: ipxe.efi:/usr/share/ipxe/ipxe-snponly-x86_64.efi,undionly.kpxe:/usr/share/ipxe/undionly.kpxe

`initial_grub_template`

Type string

Default `$pybasedir/drivers/modules/initial_grub_cfg.template`

On ironic-conductor node, the path to the initial grubconfiguration template for grub network boot.

redfish

`connection_attempts`

Type integer

Default `5`

Minimum Value 1

Maximum number of attempts to try to connect to Redfish

`connection_retry_interval`

Type integer

Default `4`

Minimum Value 1

Number of seconds to wait between attempts to connect to Redfish

`connection_cache_size`

Type integer

Default `1000`

Minimum Value 0

Maximum Redfish client connection cache size. Redfish driver would strive to reuse authenticated BMC connections (obtained through Redfish Session Service). This option caps the maximum number of connections to maintain. The value of *0* disables client connection caching completely.

auth_type

Type string

Default `auto`

Valid Values basic, session, auto

Redfish HTTP client authentication method.

Possible values

basic Use HTTP basic authentication

session Use HTTP session authentication

auto Try HTTP session authentication first, fall back to basic HTTP authentication

use_swift

Type boolean

Default `True`

Mutable This option can be changed without restarting.

Upload generated ISO images for virtual media boot to Swift, then pass temporary URL to BMC for booting the node. If set to false, images are placed on the ironic-conductor node and served over its local HTTP server.

swift_container

Type string

Default `ironic_redfish_container`

Mutable This option can be changed without restarting.

The Swift container to store Redfish driver data. Applies only when *use_swift* is enabled.

swift_object_expiry_timeout

Type integer

Default `900`

Mutable This option can be changed without restarting.

Amount of time in seconds for Swift objects to auto-expire. Applies only when *use_swift* is enabled.

kernel_append_params

Type string

Default `nofb nomodeset vga=normal`

Mutable This option can be changed without restarting.

Additional kernel parameters to pass down to the instance kernel. These parameters can be consumed by the kernel or by the applications by reading /proc/cmdline. Mind severe cmdline size limit! Can be overridden by *instance_info/kernel_append_params* property.

`file_permission`

Type integer

Default `420`

File permission for swift-less image hosting with the octal permission representation of file access permissions. This setting defaults to `644`, or as the octal number `0o644` in Python. This setting must be set to the octal number representation, meaning starting with `0o`.

`firmware_update_status_interval`

Type integer

Default `60`

Minimum Value 0

Number of seconds to wait between checking for completed firmware update tasks

`firmware_update_fail_interval`

Type integer

Default `60`

Minimum Value 0

Number of seconds to wait between checking for failed firmware update tasks

`firmware_source`

Type string

Default `http`

Valid Values http, local, swift

Mutable This option can be changed without restarting.

Specifies how firmware image should be served. Whether from its original location using the firmware source URL directly, or should serve it from ironics Swift or HTTP server.

Possible values

http If firmware source URL is also HTTP, then serve from original location, otherwise copy to ironics HTTP server. Default.

local Download from original location and server from ironics HTTP server.

swift If firmware source URL is also Swift, serve from original location, otherwise copy to ironics Swift server.

`raid_config_status_interval`

Type integer

Default `60`

Minimum Value 0

Number of seconds to wait between checking for completed raid config tasks

`raid_config_fail_interval`

Type integer

Default `60`

Minimum Value 0

Number of seconds to wait between checking for failed raid config tasks

service_catalog

`auth_url`

Type unknown type

Default `<None>`

Authentication URL

`auth_type`

Type unknown type

Default `<None>`

Authentication type to load

Table 80: Deprecated Variations

Group	Name
service_catalog	auth_plugin

`cafile`

Type string

Default <None>

PEM encoded Certificate Authority to use when verifying HTTPs connections.

`certfile`

Type string

Default <None>

PEM encoded client certificate cert file

`collect_timing`

Type boolean

Default False

Collect per-API call timing information.

`connect_retries`

Type integer

Default <None>

The maximum number of retries that should be attempted for connection errors.

`connect_retry_delay`

Type floating point

Default <None>

Delay (in seconds) between two retries for connection errors. If not set, exponential retry starting with 0.5 seconds up to a maximum of 60 seconds is used.

`default_domain_id`

Type unknown type

Default <None>

Optional domain ID to use with v3 and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

`default_domain_name`

Type unknown type

Default <None>

Optional domain name to use with v3 API and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

`domain_id`

Type unknown type

Default <None>

Domain ID to scope to

domain_name

Type unknown type

Default <None>

Domain name to scope to

endpoint_override

Type string

Default <None>

Always use this endpoint URL for requests for this client. NOTE: The unversioned endpoint should be specified here; to request a particular API version, use the *version*, *min-version*, and/or *max-version* options.

insecure

Type boolean

Default False

Verify HTTPS connections.

keyfile

Type string

Default <None>

PEM encoded client certificate key file

max_version

Type string

Default <None>

The maximum major version of a given API, intended to be used as the upper bound of a range with min_version. Mutually exclusive with version.

min_version

Type string

Default <None>

The minimum major version of a given API, intended to be used as the lower bound of a range with max_version. Mutually exclusive with version. If min_version is given with no max_version it is as if max version is latest.

password

Type unknown type

Default <None>

Users password

project_domain_id

Type unknown type

Default <None>

Domain ID containing project

project_domain_name

Type unknown type

Default <None>

Domain name containing project

project_id

Type unknown type

Default <None>

Project ID to scope to

Table 81: Deprecated Variations

Group	Name
service_catalog	tenant-id
service_catalog	tenant_id

project_name

Type unknown type

Default <None>

Project name to scope to

Table 82: Deprecated Variations

Group	Name
service_catalog	tenant-name
service_catalog	tenant_name

region_name

Type string

Default <None>

The default region_name for endpoint URL discovery.

service_name

Type string

Default <None>

The default service_name for endpoint URL discovery.

service_type

Type string

Default baremetal

The default service_type for endpoint URL discovery.

split_loggers

Type boolean

Default False

Log requests to multiple loggers.

status_code_retries

Type integer

Default <None>

The maximum number of retries that should be attempted for retriable HTTP status codes.

status_code_retry_delay

Type floating point

Default <None>

Delay (in seconds) between two retries for retriable status codes. If not set, exponential retry starting with 0.5 seconds up to a maximum of 60 seconds is used.

system_scope

Type unknown type

Default <None>

Scope for system operations

tenant_id

Type unknown type

Default <None>

Tenant ID

tenant_name

Type unknown type

Default <None>

Tenant Name

timeout

Type integer

Default <None>

Timeout value for http requests

trust_id

Type unknown type

Default <None>

Trust ID

user_domain_id

Type unknown type

Default <None>

Users domain id

user_domain_name

Type unknown type

Default <None>

Users domain name

user_id

Type unknown type

Default <None>

User id

username

Type unknown type

Default <None>

Username

Table 83: Deprecated Variations

Group	Name
service_catalog	user-name
service_catalog	user_name

valid_interfaces

Type list

Default `['internal', 'public']`

List of interfaces, in order of preference, for endpoint URL.

version

Type string

Default `<None>`

Minimum Major API version within a given Major API version for endpoint URL discovery. Mutually exclusive with min_version and max_version

snmp

power_timeout

Type integer

Default `10`

Seconds to wait for power action to be completed

reboot_delay

Type integer

Default `0`

Minimum Value 0

Time (in seconds) to sleep between when rebooting (powering off and on again)

power_action_delay

Type integer

Default `0`

Minimum Value 0

Time (in seconds) to sleep before power on and after powering off. Which may be needed with some PDUs as they may not honor toggling a specific power port in rapid succession without a delay. This option may be useful if the attached physical machine has a substantial power supply to hold it over in the event of a brownout.

udp_transport_timeout

Type floating point

Default `1.0`

Minimum Value 0.0

Response timeout in seconds used for UDP transport. Timeout should be a multiple of 0.5 seconds and is applicable to each retry.

udp_transport_retries

Type integer

Default `5`

Minimum Value 0

Maximum number of UDP request retries, 0 means no retries.

ssl

ca_file

Type string

Default `<None>`

CA certificate file to use to verify connecting clients.

Table 84: Deprecated Variations

Group	Name
DEFAULT	ssl_ca_file

cert_file

Type string

Default `<None>`

Certificate file to use when starting the server securely.

Table 85: Deprecated Variations

Group	Name
DEFAULT	ssl_cert_file

key_file

Type string

Default `<None>`

Private key file to use when starting the server securely.

Table 86: Deprecated Variations

Group	Name
DEFAULT	ssl_key_file

version

Type string

Default <None>

SSL version to use (valid only if SSL enabled). Valid values are TLSv1 and SSLv23. SSLv2, SSLv3, TLSv1_1, and TLSv1_2 may be available on some distributions.

ciphers

Type string

Default <None>

Sets the list of available ciphers. value should be a string in the OpenSSL cipher list format.

swift

auth_url

Type unknown type

Default <None>

Authentication URL

auth_type

Type unknown type

Default <None>

Authentication type to load

Table 87: Deprecated Variations

Group	Name
swift	auth_plugin

cafile

Type string

Default <None>

PEM encoded Certificate Authority to use when verifying HTTPs connections.

`certfile`

Type string

Default <None>

PEM encoded client certificate cert file

`collect_timing`

Type boolean

Default False

Collect per-API call timing information.

`connect_retries`

Type integer

Default <None>

The maximum number of retries that should be attempted for connection errors.

`connect_retry_delay`

Type floating point

Default <None>

Delay (in seconds) between two retries for connection errors. If not set, exponential retry starting with 0.5 seconds up to a maximum of 60 seconds is used.

`default_domain_id`

Type unknown type

Default <None>

Optional domain ID to use with v3 and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

`default_domain_name`

Type unknown type

Default <None>

Optional domain name to use with v3 API and v2 parameters. It will be used for both the user and project domain in v3 and ignored in v2 authentication.

`domain_id`

Type unknown type

Default <None>

Domain ID to scope to

domain_name

Type unknown type

Default <None>

Domain name to scope to

endpoint_override

Type string

Default <None>

Always use this endpoint URL for requests for this client. NOTE: The unversioned endpoint should be specified here; to request a particular API version, use the *version*, *min-version*, and/or *max-version* options.

insecure

Type boolean

Default False

Verify HTTPS connections.

keyfile

Type string

Default <None>

PEM encoded client certificate key file

max_version

Type string

Default <None>

The maximum major version of a given API, intended to be used as the upper bound of a range with min_version. Mutually exclusive with version.

min_version

Type string

Default <None>

The minimum major version of a given API, intended to be used as the lower bound of a range with max_version. Mutually exclusive with version. If min_version is given with no max_version it is as if max version is latest.

password

Type unknown type

Default <None>

Users password

project_domain_id

Type unknown type

Default <None>

Domain ID containing project

project_domain_name

Type unknown type

Default <None>

Domain name containing project

project_id

Type unknown type

Default <None>

Project ID to scope to

Table 88: Deprecated Variations

Group	Name
swift	tenant-id
swift	tenant_id

project_name

Type unknown type

Default <None>

Project name to scope to

Table 89: Deprecated Variations

Group	Name
swift	tenant-name
swift	tenant_name

region_name

Type string

Default <None>

The default region_name for endpoint URL discovery.

service_name

Type string

Default <None>

The default service_name for endpoint URL discovery.

`service_type`

Type string

Default `object-store`

The default service_type for endpoint URL discovery.

`split_loggers`

Type boolean

Default `False`

Log requests to multiple loggers.

`status_code_retries`

Type integer

Default <None>

The maximum number of retries that should be attempted for retriable HTTP status codes.

`status_code_retry_delay`

Type floating point

Default <None>

Delay (in seconds) between two retries for retriable status codes. If not set, exponential retry starting with 0.5 seconds up to a maximum of 60 seconds is used.

`swift_max_retries`

Type integer

Default `2`

Maximum number of times to retry a Swift request, before failing.

`system_scope`

Type unknown type

Default <None>

Scope for system operations

`tenant_id`

Type unknown type

Default <None>

Tenant ID

tenant_name

Type unknown type

Default <None>

Tenant Name

timeout

Type integer

Default <None>

Timeout value for http requests

trust_id

Type unknown type

Default <None>

Trust ID

user_domain_id

Type unknown type

Default <None>

Users domain id

user_domain_name

Type unknown type

Default <None>

Users domain name

user_id

Type unknown type

Default <None>

User id

username

Type unknown type

Default <None>

Username

Table 90: Deprecated Variations

Group	Name
swift	user-name
swift	user_name

valid_interfaces

Type list

Default `['internal', 'public']`

List of interfaces, in order of preference, for endpoint URL.

version

Type string

Default <None>

Minimum Major API version within a given Major API version for endpoint URL discovery. Mutually exclusive with min_version and max_version

xclarity

manager_ip

Type string

Default <None>

IP address of the XClarity Controller. Configuration here is deprecated and will be removed in the Stein release. Please update the driver_info field to use xclarity_manager_ip instead

username

Type string

Default <None>

Username for the XClarity Controller. Configuration here is deprecated and will be removed in the Stein release. Please update the driver_info field to use xclarity_username instead

password

Type string

Default <None>

Password for XClarity Controller username. Configuration here is deprecated and will be removed in the Stein release. Please update the driver_info field to use xclarity_password instead

port

Type port number

Default 443

Minimum Value 0

Maximum Value 65535

Port to be used for XClarity Controller connection.

5.1.2 Policies

Warning: JSON formatted policy files were deprecated in the Wallaby development cycle due to the Victoria deprecation by the `olso.policy` library. Use the oslopolicy-convert-json-to-yaml tool to convert the existing JSON to YAML formatted policy file in backward compatible way.

The following is an overview of all available policies in Ironic. For a sample configuration file, refer to *Ironic Policy*.

ironic.api

admin_api

Default `role:admin or role:administrator`

Legacy rule for cloud admin access

public_api

Default `is_public_api:True`

Internal flag for public API routes

show_password

Default `!`

Show or mask secrets within node driver information in API responses

show_instance_secrets

Default `!`

Show or mask secrets within instance information in API responses

is_member

Default `(project_domain_id:default or project_domain_id:None) and (project_name:demo or project_name:baremetal)`

May be used to restrict access to specific projects

is_observer

Default `rule:is_member and (role:observer or role:baremetal_observer)`

Read-only API access

`is_admin`

Default `rule:admin_api or (rule:is_member and role:baremetal_admin)`

Full read/write API access

`is_node_owner`

Default `project_id:%(node.owner)s`

Owner of node

`is_node_lessee`

Default `project_id:%(node.lessee)s`

Lessee of node

`is_allocation_owner`

Default `project_id:%(allocation.owner)s`

Owner of allocation

`baremetal:node:create`

Default `role:admin and system_scope:all`

Operations

- **POST** `/nodes`

Scope Types

- **system**

Create Node records

`baremetal:node:list`

Default `role:reader`

Operations

- **GET** `/nodes`
- **GET** `/nodes/detail`

Scope Types

- **system**
- **project**

Retrieve multiple Node records, filtered by an explicit owner or the client project_id

`baremetal:node:list_all`

Default `role:reader and system_scope:all`

Operations

- **GET** `/nodes`
- **GET** `/nodes/detail`

Scope Types

- **system**

Retrieve multiple Node records

`baremetal:node:get`

Default `(role:reader and system_scope:all) or (role:reader and (project_id:%(node.owner)s or project_id:%(node.lessee)s))`

Operations

- **GET** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Retrieve a single Node record

`baremetal:node:get:filter_threshold`

Default `role:reader and system_scope:all`

Operations

- **GET** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Filter to allow operators to govern the threshold where information should be filtered. Non-authorized users will be subjected to additional API policy checks for API content response bodies.

`baremetal:node:get:last_error`

Default `(role:reader and system_scope:all) or (role:reader and project_id:%(node.owner)s)`

Operations

- **GET** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Governs if the node last_error field is masked from APIclients with insufficent privileges.

`baremetal:node:get:reservation`

Default `(role:reader and system_scope:all) or (role:reader and project_id:%(node.owner)s)`

Operations

- **GET** `/nodes/{node_ident}`

Scope Types

- **system**

- **project**

Governs if the node reservation field is masked from APIclients with insufficent privileges.

`baremetal:node:get:driver_internal_info`

Default `(role:reader and system_scope:all) or (role:reader and project_id:%(node.owner)s)`

Operations

- **GET** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Governs if the node driver_internal_info field is masked from API clients with insufficent privileges.

`baremetal:node:get:driver_info`

Default `(role:reader and system_scope:all) or (role:reader and project_id:%(node.owner)s)`

Operations

- **GET** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Governs if the driver_info field is masked from APIclients with insufficent privileges.

`baremetal:node:update:driver_info`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s)`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Governs if node driver_info field can be updated via the API clients.

`baremetal:node:update:properties`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s)`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Governs if node properties field can be updated via the API clients.

`baremetal:node:update:chassis_uuid`

Default `role:admin and system_scope:all`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Governs if node chassis_uuid field can be updated via the API clients.

`baremetal:node:update:instance_uuid`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s)`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Governs if node instance_uuid field can be updated via the API clients.

`baremetal:node:update:lessee`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s)`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Governs if node lessee field can be updated via the API clients.

`baremetal:node:update:owner`

Default `role:member and system_scope:all`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**

- **project**

Governs if node owner field can be updated via the API clients.

`baremetal:node:update:driver_interfaces`

Default `(role:member and system_scope:all) or (role:admin and project_id:%(node.owner)s)`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Governs if node driver and driver interfaces field can be updated via the API clients.

`baremetal:node:update:network_data`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s)`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Governs if node driver_info field can be updated via the API clients.

`baremetal:node:update:conductor_group`

Default `role:member and system_scope:all`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Governs if node conductor_group field can be updated via the API clients.

`baremetal:node:update:name`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s)`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**

- **project**

Governs if node name field can be updated via the API clients.

`baremetal:node:update:retired`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s)`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Governs if node retired and retired reason can be updated by API clients.

`baremetal:node:update`

Default `(role:member and system_scope:all) or (role:member and (project_id:%(node.owner)s or project_id:%(node.lessee)s))`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Generalized update of node records

`baremetal:node:update_extra`

Default `(role:member and system_scope:all) or (role:member and (project_id:%(node.owner)s or project_id:%(node.lessee)s))`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Update Node extra field

`baremetal:node:update_instance_info`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s) or (role:admin and project_id:%(node.lessee)s)`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Update Node instance_info field

`baremetal:node:update_owner_provisioned`

Default `role:admin and system_scope:all`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**

Update Node owner even when Node is provisioned

`baremetal:node:delete`

Default `role:admin and system_scope:all`

Operations

- **DELETE** `/nodes/{node_ident}`

Scope Types

- **system**
- **project**

Delete Node records

`baremetal:node:validate`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s) or (role:admin and project_id:%(node.lessee)s)`

Operations

- **GET** `/nodes/{node_ident}/validate`

Scope Types

- **system**
- **project**

Request active validation of Nodes

`baremetal:node:set_maintenance`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s) or (role:admin and project_id:%(node.lessee)s)`

Operations

- **PUT** `/nodes/{node_ident}/maintenance`

Scope Types

- **system**

- **project**

Set maintenance flag, taking a Node out of service

`baremetal:node:clear_maintenance`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s) or (role:admin and project_id:%(node.lessee)s)`

Operations

- **DELETE** `/nodes/{node_ident}/maintenance`

Scope Types

- **system**
- **project**

Clear maintenance flag, placing the Node into service again

`baremetal:node:get_boot_device`

Default `(role:member and system_scope:all) or (role:admin and project_id:%(node.owner)s)`

Operations

- **GET** `/nodes/{node_ident}/management/boot_device`
- **GET** `/nodes/{node_ident}/management/boot_device/supported`

Scope Types

- **system**
- **project**

Retrieve Node boot device metadata

`baremetal:node:set_boot_device`

Default `(role:member and system_scope:all) or (role:admin and project_id:%(node.owner)s)`

Operations

- **PUT** `/nodes/{node_ident}/management/boot_device`

Scope Types

- **system**
- **project**

Change Node boot device

`baremetal:node:get_indicator_state`

Default `(role:reader and system_scope:all) or (role:reader and (project_id:%(node.owner)s or project_id:%(node.lessee)s))`

Operations

- **GET** /nodes/{node_ident}/management/indicators/{component}/{indicator}
- **GET** /nodes/{node_ident}/management/indicators

Scope Types

- **system**
- **project**

Retrieve Node indicators and their states

`baremetal:node:set_indicator_state`

Default (role:member and system_scope:all) or (role:member and project_id:%(node.owner)s)

Operations

- **PUT** /nodes/{node_ident}/management/indicators/{component}/{indicator}

Scope Types

- **system**
- **project**

Change Node indicator state

`baremetal:node:inject_nmi`

Default (role:member and system_scope:all) or (role:admin and project_id:%(node.owner)s)

Operations

- **PUT** /nodes/{node_ident}/management/inject_nmi

Scope Types

- **system**
- **project**

Inject NMI for a node

`baremetal:node:get_states`

Default (role:reader and system_scope:all) or (role:reader and (project_id:%(node.owner)s or project_id:%(node.lessee)s))

Operations

- **GET** /nodes/{node_ident}/states

Scope Types

- **system**
- **project**

View Node power and provision state

`baremetal:node:set_power_state`

Default `(role:member and system_scope:all) or (role:member and (project_id:%(node.owner)s or project_id:%(node.lessee)s))`

Operations

- **PUT** `/nodes/{node_ident}/states/power`

Scope Types

- **system**
- **project**

Change Node power status

`baremetal:node:set_boot_mode`

Default `(role:member and system_scope:all) or (role:member and (project_id:%(node.owner)s or project_id:%(node.lessee)s))`

Operations

- **PUT** `/nodes/{node_ident}/states/boot_mode`

Scope Types

- **system**
- **project**

Change Node boot mode

`baremetal:node:set_secure_boot`

Default `(role:member and system_scope:all) or (role:member and (project_id:%(node.owner)s or project_id:%(node.lessee)s))`

Operations

- **PUT** `/nodes/{node_ident}/states/secure_boot`

Scope Types

- **system**
- **project**

Change Node secure boot state

`baremetal:node:set_provision_state`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s) or (role:admin and project_id:%(node.lessee)s)`

Operations

- **PUT** `/nodes/{node_ident}/states/provision`

Scope Types

- **system**
- **project**

Change Node provision status

`baremetal:node:set_raid_state`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s)`

Operations

- **PUT** `/nodes/{node_ident}/states/raid`

Scope Types

- **system**
- **project**

Change Node RAID status

`baremetal:node:get_console`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s)`

Operations

- **GET** `/nodes/{node_ident}/states/console`

Scope Types

- **system**
- **project**

Get Node console connection information

`baremetal:node:set_console_state`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s)`

Operations

- **PUT** `/nodes/{node_ident}/states/console`

Scope Types

- **system**
- **project**

Change Node console status

`baremetal:node:vif:list`

Default `(role:reader and system_scope:all) or (role:reader and (project_id:%(node.owner)s or project_id:%(node.lessee)s))`

Operations

- **GET** `/nodes/{node_ident}/vifs`

Scope Types

- **system**
- **project**

List VIFs attached to node

`baremetal:node:vif:attach`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s) or (role:admin and project_id:%(node.lessee)s)`

Operations

- **POST** `/nodes/{node_ident}/vifs`

Scope Types

- **system**
- **project**

Attach a VIF to a node

`baremetal:node:vif:detach`

Default `(role:member and system_scope:all) or (role:member and project_id:%(node.owner)s) or (role:admin and project_id:%(node.lessee)s)`

Operations

- **DELETE** `/nodes/{node_ident}/vifs/{node_vif_ident}`

Scope Types

- **system**
- **project**

Detach a VIF from a node

`baremetal:node:traits:list`

Default `(role:reader and system_scope:all) or (role:reader and (project_id:%(node.owner)s or project_id:%(node.lessee)s))`

Operations

- **GET** `/nodes/{node_ident}/traits`

Scope Types

- **system**
- **project**

List node traits

`baremetal:node:traits:set`

Default `(role:member and system_scope:all) or (role:admin and project_id:%(node.owner)s)`

Operations

- **PUT** `/nodes/{node_ident}/traits`
- **PUT** `/nodes/{node_ident}/traits/{trait}`

Scope Types

- **system**
- **project**

Add a trait to, or replace all traits of, a node

`baremetal:node:traits:delete`

Default `(role:member and system_scope:all) or (role:admin and project_id:%(node.owner)s)`

Operations

- **DELETE** `/nodes/{node_ident}/traits`
- **DELETE** `/nodes/{node_ident}/traits/{trait}`

Scope Types

- **system**
- **project**

Remove one or all traits from a node

`baremetal:node:bios:get`

Default `(role:reader and system_scope:all) or (role:reader and (project_id:%(node.owner)s or project_id:%(node.lessee)s))`

Operations

- **GET** `/nodes/{node_ident}/bios`
- **GET** `/nodes/{node_ident}/bios/{setting}`

Scope Types

- **system**
- **project**

Retrieve Node BIOS information

`baremetal:node:disable_cleaning`

Default `role:admin and system_scope:all`

Operations

- **PATCH** `/nodes/{node_ident}`

Scope Types

- **system**

Disable Node disk cleaning

`baremetal:node:history:get`

Default `(role:reader and system_scope:all) or (role:reader and project_id:%(node.owner)s)`

Operations

- **GET** `/nodes/{node_ident}/history`

- **GET** /nodes/{node_ident}/history/{event_ident}

Scope Types

- **system**
- **project**

Filter to allow operators to retreive history records for a node.

`baremetal:port:get`

Default (role:reader and system_scope:all) or (role:reader and (project_id:%(node.owner)s or project_id:%(node.lessee)s))

Operations

- **GET** /ports/{port_id}
- **GET** /nodes/{node_ident}/ports
- **GET** /nodes/{node_ident}/ports/detail
- **GET** /portgroups/{portgroup_ident}/ports
- **GET** /portgroups/{portgroup_ident}/ports/detail

Scope Types

- **system**
- **project**

Retrieve Port records

`baremetal:port:list`

Default role:reader

Operations

- **GET** /ports
- **GET** /ports/detail

Scope Types

- **system**
- **project**

Retrieve multiple Port records, filtered by owner

`baremetal:port:list_all`

Default role:reader and system_scope:all

Operations

- **GET** /ports
- **GET** /ports/detail

Scope Types

- **system**
- **project**

Retrieve multiple Port records

`baremetal:port:create`

Default `(role:admin and system_scope:all) or (role:admin and project_id:%(node.owner)s)`

Operations

- **POST** `/ports`

Scope Types

- **system**
- **project**

Create Port records

`baremetal:port:delete`

Default `(role:admin and system_scope:all) or (role:admin and project_id:%(node.owner)s)`

Operations

- **DELETE** `/ports/{port_id}`

Scope Types

- **system**
- **project**

Delete Port records

`baremetal:port:update`

Default `(role:member and system_scope:all) or (role:admin and project_id:%(node.owner)s)`

Operations

- **PATCH** `/ports/{port_id}`

Scope Types

- **system**
- **project**

Update Port records

`baremetal:portgroup:get`

Default `(role:reader and system_scope:all) or (role:reader and (project_id:%(node.owner)s or project_id:%(node.lessee)s))`

Operations

- **GET** `/portgroups`
- **GET** `/portgroups/detail`
- **GET** `/portgroups/{portgroup_ident}`
- **GET** `/nodes/{node_ident}/portgroups`

- **GET** /nodes/{node_ident}/portgroups/detail

Scope Types

- **system**
- **project**

Retrieve Portgroup records

`baremetal:portgroup:create`

Default (role:admin and system_scope:all) or (role:admin and project_id:%(node.owner)s)

Operations

- **POST** /portgroups

Scope Types

- **system**
- **project**

Create Portgroup records

`baremetal:portgroup:delete`

Default (role:admin and system_scope:all) or (role:admin and project_id:%(node.owner)s)

Operations

- **DELETE** /portgroups/{portgroup_ident}

Scope Types

- **system**
- **project**

Delete Portgroup records

`baremetal:portgroup:update`

Default (role:member and system_scope:all) or (role:admin and project_id:%(node.owner)s)

Operations

- **PATCH** /portgroups/{portgroup_ident}

Scope Types

- **system**
- **project**

Update Portgroup records

`baremetal:portgroup:list`

Default role:reader

Operations

- **GET** /portgroups
- **GET** /portgroups/detail

Scope Types

- **system**
- **project**

Retrieve multiple Port records, filtered by owner

`baremetal:portgroup:list_all`

Default role:reader and system_scope:all

Operations

- **GET** /portgroups
- **GET** /portgroups/detail

Scope Types

- **system**
- **project**

Retrieve multiple Port records

`baremetal:chassis:get`

Default role:reader and system_scope:all

Operations

- **GET** /chassis
- **GET** /chassis/detail
- **GET** /chassis/{chassis_id}

Scope Types

- **system**

Retrieve Chassis records

`baremetal:chassis:create`

Default role:admin and system_scope:all

Operations

- **POST** /chassis

Scope Types

- **system**

Create Chassis records

`baremetal:chassis:delete`

Default role:admin and system_scope:all

Operations

- **DELETE** /chassis/{chassis_id}

Scope Types

- **system**

Delete Chassis records

`baremetal:chassis:update`

Default role:member and system_scope:all

Operations

- **PATCH** /chassis/{chassis_id}

Scope Types

- **system**

Update Chassis records

`baremetal:driver:get`

Default role:reader and system_scope:all

Operations

- **GET** /drivers
- **GET** /drivers/{driver_name}

Scope Types

- **system**

View list of available drivers

`baremetal:driver:get_properties`

Default role:reader and system_scope:all

Operations

- **GET** /drivers/{driver_name}/properties

Scope Types

- **system**

View driver-specific properties

`baremetal:driver:get_raid_logical_disk_properties`

Default role:reader and system_scope:all

Operations

- **GET** /drivers/{driver_name}/raid/logical_disk_properties

Scope Types

- **system**

View driver-specific RAID metadata

`baremetal:node:vendor_passthru`

Default `role:admin and system_scope:all`

Operations

- **GET** `nodes/{node_ident}/vendor_passthru/methods`
- **GET** `nodes/{node_ident}/vendor_passthru?method={method_name}`
- **PUT** `nodes/{node_ident}/vendor_passthru?method={method_name}`
- **POST** `nodes/{node_ident}/vendor_passthru?method={method_name}`
- **PATCH** `nodes/{node_ident}/vendor_passthru?method={method_name}`
- **DELETE** `nodes/{node_ident}/vendor_passthru?method={method_name}`

Scope Types

- **system**
- **project**

Access vendor-specific Node functions

`baremetal:driver:vendor_passthru`

Default `role:admin and system_scope:all`

Operations

- **GET** `drivers/{driver_name}/vendor_passthru/methods`
- **GET** `drivers/{driver_name}/vendor_passthru?method={method_name}`
- **PUT** `drivers/{driver_name}/vendor_passthru?method={method_name}`
- **POST** `drivers/{driver_name}/vendor_passthru?method={method_name}`
- **PATCH** `drivers/{driver_name}/vendor_passthru?method={method_name}`
- **DELETE** `drivers/{driver_name}/vendor_passthru?method={method_name}`

Scope Types

- **system**

Access vendor-specific Driver functions

`baremetal:node:ipa_heartbeat`

Default <empty string>

Operations

- **POST** `/heartbeat/{node_ident}`

Receive heartbeats from IPA ramdisk

`baremetal:driver:ipa_lookup`

Default <empty string>

Operations

- **GET** `/lookup`

Access IPA ramdisk functions

`baremetal:volume:list_all`

Default `role:reader and system_scope:all`

Operations

- **GET** `/volume/connectors`
- **GET** `/volume/targets`
- **GET** `/nodes/{node_ident}/volume/connectors`
- **GET** `/nodes/{node_ident}/volume/targets`

Scope Types

- **system**
- **project**

Retrieve a list of all Volume connector and target records

`baremetal:volume:list`

Default `role:reader`

Operations

- **GET** `/volume/connectors`
- **GET** `/volume/targets`
- **GET** `/nodes/{node_ident}/volume/connectors`
- **GET** `/nodes/{node_ident}/volume/targets`

Scope Types

- **system**
- **project**

Retrieve a list of Volume connector and target records

`baremetal:volume:get`

Default `(role:reader and system_scope:all) or (role:reader and (project_id:%(node.owner)s or project_id:%(node.lessee)s))`

Operations

- **GET** `/volume`
- **GET** `/volume/connectors`
- **GET** `/volume/connectors/{volume_connector_id}`

- **GET** /volume/targets
- **GET** /volume/targets/{volume_target_id}
- **GET** /nodes/{node_ident}/volume
- **GET** /nodes/{node_ident}/volume/connectors
- **GET** /nodes/{node_ident}/volume/targets

Scope Types

- **system**
- **project**

Retrieve Volume connector and target records

`baremetal:volume:create`

Default (role:member and system_scope:all) or (role:admin and project_id:%(node.owner)s) or (role:admin and project_id:%(node.lessee)s)

Operations

- **POST** /volume/connectors
- **POST** /volume/targets

Scope Types

- **system**
- **project**

Create Volume connector and target records

`baremetal:volume:delete`

Default (role:member and system_scope:all) or (role:admin and project_id:%(node.owner)s) or (role:admin and project_id:%(node.lessee)s)

Operations

- **DELETE** /volume/connectors/{volume_connector_id}
- **DELETE** /volume/targets/{volume_target_id}

Scope Types

- **system**
- **project**

Delete Volume connector and target records

`baremetal:volume:update`

Default (role:member and system_scope:all) or (role:member and project_id:%(node.owner)s) or (role:admin and project_id:%(node.lessee)s)

Operations

- **PATCH** /volume/connectors/{volume_connector_id}
- **PATCH** /volume/targets/{volume_target_id}

Scope Types

- **system**
- **project**

Update Volume connector and target records

`baremetal:volume:view_target_properties`

Default (role:reader and system_scope:all) or (role:admin)

Operations

- **GET** /volume/connectors/{volume_connector_id}
- **GET** /volume/targets/{volume_target_id}

Scope Types

- **system**
- **project**

Ability to view volume target properties

`baremetal:conductor:get`

Default role:reader and system_scope:all

Operations

- **GET** /conductors
- **GET** /conductors/{hostname}

Scope Types

- **system**

Retrieve Conductor records

`baremetal:allocation:get`

Default (role:reader and system_scope:all) or (role:reader and project_id:%(allocation.owner)s)

Operations

- **GET** /allocations/{allocation_id}
- **GET** /nodes/{node_ident}/allocation

Scope Types

- **system**
- **project**

Retrieve Allocation records

`baremetal:allocation:list`

Default role:reader

Operations

- **GET** /allocations

Scope Types

- **system**
- **project**

Retrieve multiple Allocation records, filtered by owner

`baremetal:allocation:list_all`

Default role:reader and system_scope:all

Operations

- **GET** /allocations

Scope Types

- **system**
- **project**

Retrieve multiple Allocation records

`baremetal:allocation:create`

Default (role:member and system_scope:all) or (role:member)

Operations

- **POST** /allocations

Scope Types

- **system**
- **project**

Create Allocation records

`baremetal:allocation:create_restricted`

Default role:member and system_scope:all

Operations

- **POST** /allocations

Scope Types

- **system**
- **project**

Create Allocation records with a specific owner.

`baremetal:allocation:delete`

Default (role:member and system_scope:all) or (role:member and project_id:%(allocation.owner)s)

Operations

- **DELETE** /allocations/{allocation_id}

- **DELETE** /nodes/{node_ident}/allocation

Scope Types

- **system**
- **project**

Delete Allocation records

`baremetal:allocation:update`

Default (role:member and system_scope:all) or (role:member and project_id:%(allocation.owner)s)

Operations

- **PATCH** /allocations/{allocation_id}

Scope Types

- **system**
- **project**

Change name and extra fields of an allocation

`baremetal:allocation:create_pre_rbac`

Default (rule:is_member and role:baremetal_admin) or (is_admin_project:True and role:admin)

Operations

- **PATCH** /allocations/{allocation_id}

Scope Types

- **project**

Logical restrictor to prevent legacy allocation rule missuse - Requires blank allocations to originate from the legacy baremetal_admin.

`baremetal:events:post`

Default role:admin and system_scope:all

Operations

- **POST** /events

Scope Types

- **system**

Post events

`baremetal:deploy_template:get`

Default role:reader and system_scope:all

Operations

- **GET** /deploy_templates
- **GET** /deploy_templates/{deploy_template_ident}

Scope Types

- **system**

Retrieve Deploy Template records

`baremetal:deploy_template:create`

Default `role:admin and system_scope:all`

Operations

- **POST** `/deploy_templates`

Scope Types

- **system**

Create Deploy Template records

`baremetal:deploy_template:delete`

Default `role:admin and system_scope:all`

Operations

- **DELETE** `/deploy_templates/{deploy_template_ident}`

Scope Types

- **system**

Delete Deploy Template records

`baremetal:deploy_template:update`

Default `role:admin and system_scope:all`

Operations

- **PATCH** `/deploy_templates/{deploy_template_ident}`

Scope Types

- **system**

Update Deploy Template records

CHAPTER

SIX

BARE METAL API REFERENCES

Ironics REST API has changed since its first release, and continues to evolve to meet the changing needs of the community. Here we provide a conceptual guide as well as more detailed reference documentation.

6.1 REST API Conceptual Guide

6.1.1 Versioning

The ironic REST API supports two types of versioning:

- major versions, which have dedicated urls.
- microversions, which can be requested through the use of the `X-OpenStack-Ironic-API-Version` header.

There is only one major version supported currently, v1. As such, most URLs in this documentation are written with the /v1/ prefix.

Starting with the Kilo release, ironic supports microversions. In this context, a version is defined as a string of 2 integers separated by a dot: **X.Y**. Here `X` is a major version, always equal to `1`, and `Y` is a minor version. Server minor version is increased every time the API behavior is changed (note *Exceptions from Versioning*).

Note: Nova versioning documentation has a nice guide for developers on when to bump an API version.

The server indicates its minimum and maximum supported API versions in the `X-OpenStack-Ironic-API-Minimum-Version` and `X-OpenStack-Ironic-API-Maximum-Version` headers respectively, returned with every response. Client may request a specific API version by providing `X-OpenStack-Ironic-API-Version` header with request.

The requested microversion determines both the allowable requests and the response format for all requests. A resource may be represented differently based on the requested microversion.

If no version is requested by the client, the minimum supported version will be assumed. In this way, a client is only exposed to those API features that are supported in the requested (explicitly or implicitly) API version (again note *Exceptions from Versioning*, they are not covered by this rule).

We recommend clients that require a stable API to always request a specific version of API that they have been tested against.

Note: A special value `latest` can be requested instead a numerical microversion, which always requests the newest supported API version from the server.

REST API Versions History

REST API Version History

1.78 (Xena, 18.2)

Add endpoints to allow history events for nodes to be retrieved via the REST API.

- `GET /v1/nodes/{node_ident}/history/`
- `GET /v1/nodes/{node_ident}/history/{event_uuid}`

1.77 (Xena, 18.2)

Add a fields selector to the the Drivers list: * `GET /v1/drivers?fields=` Also add a fields selector to the the Driver detail: * `GET /v1/drivers/{driver_name}?fields=`

1.76 (Xena, 18.2)

Add endpoints for changing boot mode and secure boot state of node asynchronously:

- `PUT /v1/nodes/{node_ident}/states/boot_mode`
- `PUT /v1/nodes/{node_ident}/states/secure_boot`

1.75 (Xena, 18.1)

Add `boot_mode` and `secure_boot` to node object and expose their state at:

- `/v1/nodes/{node_ident}/states`

1.74 (Xena, 18.0)

Add support for BIOS registry fields which include details about the BIOS setting. Included in the `/v1/nodes/{node_ident}/bios/{setting}` response.

Add a new selector to include the fields in the BIOS settings list:

- `/v1/nodes/{node_ident}/bios/?detail=`

Also add a fields selector to the the BIOS settings list:

- `/v1/nodes/{node_ident}/bios/?fields=`

1.73 (Xena, 18.0)

Add a new `deploy` verb as an alias to `active` and `undeploy` verb as an alias to `deleted`.

1.72 (Wallaby, 17.0)

Add support for `agent_status` and `agent_status_message` to /v1/heartbeat. These fields are used for external installation tools, such as Anaconda, to report back status.

1.71 (Wallaby, 17.0)

Signifier of the API supporting keystone `system` scoped roles and access controls. This is an informational flag for clients to be aware of the servers capability.

1.70 (Wallaby, 17.0)

Add support for `disable_ramdisk` parameter to provisioning endpoint `/v1/nodes/{node_ident}/states/provision`.

1.69 (Wallaby, 16.2)

Add support for `deploy_steps` parameter to provisioning endpoint `/v1/nodes/{node_ident}/states/provision`. Available and optional when target is active or rebuild.

1.68 (Victoria, 16.0)

Added the `agent_verify_ca` parameter to the ramdisk heartbeat API.

1.67 (Victoria, 15.1)

Add support for the mutually exclusive `port_uuid` and `portgroup_uuid` fields by having the node vif_attach API accept those values within `vif_info`. If one is specified, then Ironic will attempt to attach a VIF to the relative port or portgroup.

1.66 (Victoria, 15.1)

Add `network_data` field to the node object, that will be used by stand-alone ironic to pass L3 network configuration information to ramdisk.

1.65 (Ussuri, 15.0)

Added `lessee` field to the node object. The field should match the `project_id` of the intended lessee. If an allocation has an owner, then the allocation process will only match the allocation with a node that has the same `owner` or `lessee`.

1.64 (Ussuri, 15.0)

Added the `network_type` to the port objects `local_link_connection` field. The `network_type` can be set to either `managed` or `unmanaged`. When the type is `unmanaged` other fields are not required. Use `unmanaged` when the neutron `network_interface` is required, but the network is in fact a flat network where no actual switch management is done.

1.63 (Ussuri, 15.0)

Added the following new endpoints for indicator management:

- `GET /v1/nodes/<node_ident>/management/indicators` to list all available indicators names for each of the hardware component. Currently known components are: `chassis`, `system`, `disk`, `power` and `nic`.
- `GET /v1/nodes/<node_ident>/management/indicators/<component>/<indicator_ident>` to retrieve all indicators and their states for the hardware component.
- `PUT /v1/nodes/<node_ident>/management/indicators/<component>/<indicator_ident>` change state of the desired indicators of the component.

1.62 (Ussuri, 15.0)

This version of the API is to signify capability of an ironic deployment to support the `agent token` functionality with the `ironic-python-agent`.

1.61 (Ussuri, 14.0)

Added `retired` field to the node object to mark nodes for retirement. If set, this flag will move nodes to `manageable` upon automatic cleaning. `manageable` nodes which have this flag set cannot be moved to available. Also added `retired_reason` to specify the retirement reason.

1.60 (Ussuri, 14.0)

Added `owner` field to the allocation object. The field should match the `project_id` of the intended owner. If the `owner` field is set, the allocation process will only match the allocation with a node that has the same `owner` field set.

1.59 (Ussuri, 14.0)

Added the ability to specify a `vendor_data` dictionary field in the `configdrive` parameter submitted with the deployment of a node. The value is a dictionary which is served as `vendor_data2.json` in the config drive.

1.58 (Train, 12.2.0)

Added the ability to backfill allocations for already deployed nodes by creating an allocation with `node` set.

1.57 (Train, 12.2.0)

Added the following new endpoint for allocation:

- `PATCH /v1/allocations/<allocation_ident>` that allows updating `name` and `extra` fields for an existing allocation.

1.56 (Stein, 12.1.0)

Added the ability for the `configdrive` parameter submitted with the deployment of a node, to include a `meta_data`, `network_data` and `user_data` dictionary fields. Ironic will now use the supplied data to create a configuration drive for the user. Prior uses of the `configdrive` field are unaffected.

1.55 (Stein, 12.1.0)

Added the following new endpoints for deploy templates:

- `GET /v1/deploy_templates` to list all deploy templates.
- `GET /v1/deploy_templates/<deploy template identifier>` to retrieve details of a deploy template.
- `POST /v1/deploy_templates` to create a deploy template.
- `PATCH /v1/deploy_templates/<deploy template identifier>` to update a deploy template.
- `DELETE /v1/deploy_templates/<deploy template identifier>` to delete a deploy template.

1.54 (Stein, 12.1.0)

Added new endpoints for external `events`:

- POST /v1/events for creating events. (This endpoint is only intended for internal consumption.)

1.53 (Stein, 12.1.0)

Added `is_smartnic` field to the port object to enable Smart NIC port creation in addition to local link connection attributes `port_id` and `hostname`.

1.52 (Stein, 12.1.0)

Added allocation API, allowing reserving a node for deployment based on resource class and traits. The new endpoints are:

- `POST /v1/allocations` to request an allocation.
- `GET /v1/allocations` to list all allocations.
- `GET /v1/allocations/<ID or name>` to retrieve the allocation details.
- `GET /v1/nodes/<ID or name>/allocation` to retrieve an allocation associated with the node.
- `DELETE /v1/allocations/<ID or name>` to remove the allocation.
- `DELETE /v1/nodes/<ID or name>/allocation` to remove an allocation associated with the node.

Also added a new field `allocation_uuid` to the node resource.

1.51 (Stein, 12.1.0)

Added `description` field to the node object to enable operators to store any information relates to the node. The field is limited to 4096 characters.

1.50 (Stein, 12.1.0)

Added `owner` field to the node object to enable operators to store information in relation to the owner of a node. The field is up to 255 characters and MAY be used in a later point in time to allow designation and deligation of permissions.

1.49 (Stein, 12.0.0)

Added new endpoints for retrieving conductors information, and added a `conductor` field to node object.

1.48 (Stein, 12.0.0)

Added `protected` field to the node object to allow protecting deployed nodes from undeploying, rebuilding or deletion. Also added `protected_reason` to specify the reason of making the node protected.

1.47 (Stein, 12.0.0)

Added `automated_clean` field to the node object, enabling cleaning per node.

1.46 (Rocky, 11.1.0)

Added `conductor_group` field to the node and the node response, as well as support to the API to return results by matching the parameter.

1.45 (Rocky, 11.1.0)

Added `reset_interfaces` parameter to nodes PATCH request, to specify whether to reset hardware interfaces to their defaults on drivers update.

1.44 (Rocky, 11.1.0)

Added `deploy_step` to the node object, to indicate the current deploy step (if any) being performed on the node.

1.43 (Rocky, 11.0.0)

Added `?detail=` boolean query to the API list endpoints to provide a more RESTful alternative to the existing `/nodes/detail` and similar endpoints.

1.42 (Rocky, 11.0.0)

Added `fault` to the node object, to indicate currently detected fault on the node.

1.41 (Rocky, 11.0.0)

Added support to abort inspection of a node in the `inspect wait` state.

1.40 (Rocky, 11.0.0)

Added BIOS properties as sub resources of nodes:

- GET /v1/nodes/<node_ident>/bios
- GET /v1/nodes/<node_ident>/bios/<setting_name>

Added `bios_interface` field to the node object to allow getting and setting the interface.

1.39 (Rocky, 11.0.0)

Added `inspect wait` to available provision states. A node is shown as `inspect wait` instead of `inspecting` during asynchronous inspection.

1.38 (Queens, 10.1.0)

Added provision_state verbs `rescue` and `unrescue` along with the following states: `rescue`, `rescue failed`, `rescue wait`, `rescuing`, `unrescue failed`, and `unrescuing`. After rescuing a node, it will be left in the `rescue` state running a rescue ramdisk, configured with the `rescue_password`, and listening with ssh on the specified network interfaces. Unrescuing a node will return it to `active`.

Added `rescue_interface` to the node object, to allow setting the rescue interface for a dynamic driver.

1.37 (Queens, 10.1.0)

Adds support for node traits, with the following new endpoints.

- GET /v1/nodes/<node identifier>/traits lists the traits for a node.
- PUT /v1/nodes/<node identifier>/traits sets all traits for a node.
- PUT /v1/nodes/<node identifier>/traits/<trait> adds a trait to a node.
- DELETE /v1/nodes/<node identifier>/traits removes all traits from a node.
- DELETE /v1/nodes/<node identifier>/traits/<trait> removes a trait from a node.

A nodes traits are also included the following node query and list responses:

- GET /v1/nodes/<node identifier>
- GET /v1/nodes/detail
- GET /v1/nodes?fields=traits

Traits cannot be specified on node creation, nor can they be updated via a PATCH request on the node.

1.36 (Queens, 10.0.0)

Added `agent_version` parameter to deploy heartbeat request for version negotiation with Ironic Python Agent features.

1.35 (Queens, 9.2.0)

Added ability to provide `configdrive` when node is updated to `rebuild` provision state.

1.34 (Pike, 9.0.0)

Adds a `physical_network` field to the port object. All ports in a portgroup must have the same value in their `physical_network` field.

1.33 (Pike, 9.0.0)

Added `storage_interface` field to the node object to allow getting and setting the interface.

Added `default_storage_interface` and `enabled_storage_interfaces` fields to the driver object to show the information.

1.32 (Pike, 9.0.0)

Added new endpoints for remote volume configuration:

- GET /v1/volume as a root for volume resources
- GET /v1/volume/connectors for listing volume connectors
- POST /v1/volume/connectors for creating a volume connector
- GET /v1/volume/connectors/<UUID> for showing a volume connector
- PATCH /v1/volume/connectors/<UUID> for updating a volume connector
- DELETE /v1/volume/connectors/<UUID> for deleting a volume connector
- GET /v1/volume/targets for listing volume targets
- POST /v1/volume/targets for creating a volume target
- GET /v1/volume/targets/<UUID> for showing a volume target
- PATCH /v1/volume/targets/<UUID> for updating a volume target
- DELETE /v1/volume/targets/<UUID> for deleting a volume target

Volume resources also can be listed as sub resources of nodes:

- GET /v1/nodes/<node identifier>/volume
- GET /v1/nodes/<node identifier>/volume/connectors
- GET /v1/nodes/<node identifier>/volume/targets

1.31 (Ocata, 7.0.0)

Added the following fields to the node object, to allow getting and setting interfaces for a dynamic driver:

- boot_interface
- console_interface
- deploy_interface
- inspect_interface
- management_interface
- power_interface
- raid_interface
- vendor_interface

1.30 (Ocata, 7.0.0)

Added dynamic driver APIs:

- GET /v1/drivers now accepts a `type` parameter (optional, one of `classic` or `dynamic`), to limit the result to only classic drivers or dynamic drivers (hardware types). Without this parameter, both classic and dynamic drivers are returned.
- GET /v1/drivers now accepts a `detail` parameter (optional, one of `True` or `False`), to show all fields for a driver. Defaults to `False`.
- GET /v1/drivers now returns an additional `type` field to show if the driver is classic or dynamic.
- GET /v1/drivers/<name> now returns an additional `type` field to show if the driver is classic or dynamic.
- GET /v1/drivers/<name> now returns additional fields that are null for classic drivers, and set as following for dynamic drivers:
  - The value of the default_<interface-type>_interface is the entrypoint name of the calculated default interface for that type:
    - default_boot_interface
    - default_console_interface
    - default_deploy_interface
    - default_inspect_interface
    - default_management_interface
    - default_network_interface
    - default_power_interface
    - default_raid_interface
    - default_vendor_interface
  - The value of the enabled_<interface-type>_interfaces is a list of entrypoint names of the enabled interfaces for that type:

* enabled_boot_interfaces
* enabled_console_interfaces
* enabled_deploy_interfaces
* enabled_inspect_interfaces
* enabled_management_interfaces
* enabled_network_interfaces
* enabled_power_interfaces
* enabled_raid_interfaces
* enabled_vendor_interfaces

1.29 (Ocata, 7.0.0)

Add a new management API to support inject NMI, PUT /v1/nodes/(node_ident)/management/inject_nmi.

1.28 (Ocata, 7.0.0)

Add /v1/nodes/<node identifier>/vifs endpoint for attach, detach and list of VIFs.

1.27 (Ocata, 7.0.0)

Add `soft rebooting` and `soft power off` as possible values for the `target` field of the power state change payload, and also add `timeout` field to it.

1.26 (Ocata, 7.0.0)

Add portgroup `mode` and `properties` fields.

1.25 (Ocata, 7.0.0)

Add possibility to unset chassis_uuid from a node.

1.24 (Ocata, 7.0.0)

Added new endpoints /v1/nodes/<node>/portgroups and /v1/portgroups/<portgroup>/ports. Added new field `port.portgroup_uuid`.

1.23 (Ocata, 7.0.0)

Added /v1/portgroups/ endpoint.

1.22 (Newton, 6.1.0)

Added endpoints for deployment ramdisks.

1.21 (Newton, 6.1.0)

Add node `resource_class` field.

1.20 (Newton, 6.1.0)

Add node `network_interface` field.

1.19 (Newton, 6.1.0)

Add `local_link_connection` and `pxe_enabled` fields to the port object.

1.18 (Newton, 6.1.0)

Add `internal_info` readonly field to the port object, that will be used by ironic to store internal port-related information.

1.17 (Newton, 6.0.0)

Addition of provision_state verb `adopt` which allows an operator to move a node from `manageable` state to `active` state without performing a deployment operation on the node. This is intended for nodes that have already been deployed by external means.

1.16 (Mitaka, 5.0.0)

Add ability to filter nodes by driver.

1.15 (Mitaka, 5.0.0)

Add ability to do manual cleaning when a node is in the manageable provision state via PUT v1/nodes/<identifier>/states/provision, target:clean, clean_steps:[].

1.14 (Liberty, 4.2.0)

Make the following endpoints discoverable via Ironic API:

- /v1/nodes/<UUID or logical name>/states
- /v1/drivers/<driver name>/properties

1.13 (Liberty, 4.2.0)

Add a new verb `abort` to the API used to abort nodes in `CLEANWAIT` state.

1.12 (Liberty, 4.2.0)

This API version adds the following abilities:

- Get/set `node.target_raid_config` and to get `node.raid_config`.
- Retrieve the logical disk properties for the driver.

1.11 (Liberty, 4.0.0, breaking change)

Newly registered nodes begin in the `enroll` provision state by default, instead of `available`. To get them to the `available` state, the `manage` action must first be run to verify basic hardware control. On success the node moves to `manageable` provision state. Then the `provide` action must be run. Automated cleaning of the node is done and the node is made `available`.

1.10 (Liberty, 4.0.0)

Logical node names support all RFC 3986 unreserved characters. Previously only valid fully qualified domain names could be used.

1.9 (Liberty, 4.0.0)

Add ability to filter nodes by provision state.

1.8 (Liberty, 4.0.0)

Add ability to return a subset of resource fields.

1.7 (Liberty, 4.0.0)

Add node `clean_step` field.

1.6 (Kilo)

Add *Hardware Inspection* process: introduce `inspecting` and `inspectfail` provision states, and `inspect` action that can be used when a node is in `manageable` provision state.

1.5 (Kilo)

Add logical node names that can be used to address a node in addition to the node UUID. Name is expected to be a valid fully qualified domain name in this version of API.

1.4 (Kilo)

Add `manageable` state and `manage` transition, which can be used to move a node to `manageable` state from `available`. The node cannot be deployed in `manageable` state. This change is mostly a preparation for future inspection work and introduction of `enroll` provision state.

1.3 (Kilo)

Add node `driver_internal_info` field.

1.2 (Kilo, breaking change)

Renamed NOSTATE (`None` in Python, `null` in JSON) node state to `available`. This is needed to reduce confusion around `None` state, especially when future additions to the state machine land.

1.1 (Kilo)

This was the initial version when API versioning was introduced. Includes the following changes from Kilo release cycle:

- Add node `maintenance_reason` field and an API endpoint to set/unset the node maintenance mode.
- Add sync and async support for vendor passthru methods.
- Vendor passthru endpoints support different HTTP methods, not only `POST`.
- Make vendor methods discoverable via the Ironic API.

- Add logic to store the config drive passed by Nova.

This has been the minimum supported version since versioning was introduced.

1.0 (Juno)

This version denotes Juno API and was never explicitly supported, as API versioning was not implemented in Juno, and 1.1 became the minimum supported version in Kilo.

Exceptions from Versioning

The following API-visible things are not covered by the API versioning:

- Current node state is always exposed as it is, even if not supported by the requested API version, with exception of `available` state, which is returned in version 1.1 as `None` (in Python) or `null` (in JSON).
- Data within free-form JSON attributes: `properties`, `driver_info`, `instance_info`, `driver_internal_info` fields on a node object; `extra` fields on all objects.
- Addition of new drivers.
- All vendor passthru methods.

CHAPTER

SEVEN

COMMAND REFERENCES

Here are references for commands not elsewhere documented.

7.1 Command References

Here are references for commands not elsewhere documented.

7.1.1 ironic-dbsync

The **ironic-dbsync** utility is used to create the database schema tables that the ironic services will use for storage. It can also be used to upgrade existing database tables when migrating between different versions of ironic.

The Alembic library is used to perform the database migrations.

Options

This is a partial list of the most useful options. To see the full list, run the following:

```
ironic-dbsync --help
```

-h, --help
Show help message and exit.

--config-dir <DIR>
Path to a config directory with configuration files.

--config-file <PATH>
Path to a configuration file to use.

-d, --debug
Print debugging output.

--version
Show the programs version number and exit.

upgrade, stamp, revision, version, create_schema, online_data_migrations
The *command* to run.

Usage

Options for the various *commands* for **ironic-dbsync** are listed when the *-h* or *--help* option is used after the command.

For example:

```
ironic-dbsync create_schema --help
```

Information about the database is read from the ironic configuration file used by the API server and conductor services. This file must be specified with the *--config-file* option:

```
ironic-dbsync --config-file /path/to/ironic.conf create_schema
```

The configuration file defines the database backend to use with the *connection* database option:

```
[database]
connection=mysql+pymysql://root@localhost/ironic
```

If no configuration file is specified with the *--config-file* option, **ironic-dbsync** assumes an SQLite database.

Command Options

ironic-dbsync is given a command that tells the utility what actions to perform. These commands can take arguments. Several commands are available:

create_schema

-h, --help
Show help for create_schema and exit.

This command will create database tables based on the most current version. It assumes that there are no existing tables.

An example of creating database tables with the most recent version:

```
ironic-dbsync --config-file=/etc/ironic/ironic.conf create_schema
```

online_data_migrations

-h, --help
Show help for online_data_migrations and exit.

--max-count <NUMBER>
The maximum number of objects (a positive value) to migrate. Optional. If not specified, all the objects will be migrated (in batches of 50 to avoid locking the database for long periods of time).

--option <MIGRATION.KEY=VALUE>
If a migration accepts additional parameters, they can be passed via this argument. It can be specified several times.

This command will migrate objects in the database to their most recent versions. This command must be successfully run (return code 0) before upgrading to a future release.

It returns:

- 1 (not completed) if there are still pending objects to be migrated. Before upgrading to a newer release, this command must be run until 0 is returned.
- 0 (success) after migrations are finished or there are no data to migrate
- 127 (error) if max-count is not a positive value or an option is invalid
- 2 (error) if the database is not compatible with this release. This command needs to be run using the previous release of ironic, before upgrading and running it with this release.

revision

-h, --help
Show help for revision and exit.

-m <MESSAGE>, **--message** <MESSAGE>
The message to use with the revision file.

--autogenerate
Compares table metadata in the application with the status of the database and generates migrations based on this comparison.

This command will create a new revision file. You can use the *--message* option to comment the revision.

This is really only useful for ironic developers making changes that require database changes. This revision file is used during database migration and will specify the changes that need to be made to the database tables. Further discussion is beyond the scope of this document.

stamp

-h, --help
Show help for stamp and exit.

--revision <REVISION>
The revision number.

This command will stamp the revision table with the version specified with the *--revision* option. It will not run any migrations.

upgrade

-h, --help
Show help for upgrade and exit.

--revision <REVISION>
The revision number to upgrade to.

This command will upgrade existing database tables to the most recent version, or to the version specified with the *--revision* option.

Before this upgrade is invoked, the command **ironic-dbsync online_data_migrations** must have been successfully run using the previous version of ironic (if you are doing an upgrade as opposed to a new installation of ironic). If it wasnt run, the database will not be compatible with this recent version of ironic, and this command will return 2 (error).

If there are no existing tables, then new tables are created, beginning with the oldest known version, and successively upgraded using all of the database migration files, until they are at the specified version. Note that this behavior is different from the *create_schema* command that creates the tables based on the most recent version.

An example of upgrading to the most recent table versions:

```
ironic-dbsync --config-file=/etc/ironic/ironic.conf upgrade
```

Note: This command is the default if no command is given to **ironic-dbsync**.

Warning: The upgrade command is not compatible with SQLite databases since it uses ALTER TABLE commands to upgrade the database tables. SQLite supports only a limited subset of ALTER TABLE.

version

-h, --help
Show help for version and exit.

This command will output the current database version.

7.1.2 ironic-status

Synopsis

```
ironic-status <category> <command> [<args>]
```

Description

ironic-status is a tool that provides routines for checking the status of a Ironic deployment.

Options

The standard pattern for executing a **ironic-status** command is:

```
ironic-status <category> <command> [<args>]
```

Run without arguments to see a list of available command categories:

```
ironic-status
```

Categories are:

- `upgrade`

Detailed descriptions are below.

You can also run with a category argument such as `upgrade` to see a list of all commands in that category:

```
ironic-status upgrade
```

These sections describe the available categories and arguments for **ironic-status**.

Upgrade

ironic-status upgrade check Performs a release-specific readiness check before restarting services with new code. This command expects to have complete configuration and access to databases and services.

Return Codes

Return code	Description
0	All upgrade readiness checks passed successfully and there is nothing to do.
1	At least one check encountered an issue and requires further investigation. This is considered a warning but the upgrade may be OK.
2	There was an upgrade status check failure that needs to be investigated. This should be considered something that stops an upgrade.
255	An unexpected error occurred.

History of Checks

12.0.0 (Stein)

- Adds a check for compatibility of the object versions with the release of ironic.

Wallaby

- Adds a check to validate the configured policy file is not JSON based as JSON based policies have been deprecated.

CHAPTER

EIGHT

CONTRIBUTOR GUIDE

8.1 Developers Guide

8.1.1 Getting Started

If you are new to ironic, this section contains information that should help you get started as a developer working on the project or contributing to the project.

Bare Metal Community

This document provides information on how to reach out to the community for questions, bug reports or new code contributions.

Useful Links

Bug/Task tracker https://storyboard.openstack.org/#!/project/openstack/ironic

Code Hosting https://opendev.org/openstack/ironic

Code Review https://review.opendev.org/#/q/status:open+project:openstack/ironic,n,z

Weekly Meeting Agenda https://wiki.openstack.org/wiki/Meetings/Ironic#Agenda_for_next_meeting

Asking Questions

There are two many venues where all discussions happen: IRC and mailing lists.

Internet Relay Chat IRC

Daily contributor discussions take place on IRC in the `#openstack-ironic` channel on the OFTC IRC network. Please feel free to connect to `ircs://irc.oftc.net:6697` and join our channel!

Note that while we have community members from everywhere in the world, were the most active from roughly 6am to 12am. If you dont get an answer to your question, try the *Mailing list*.

Additional information on getting connected can be found in the OpenStack community contribution guide.

Mailing list

We use the *openstack-discuss* mailing list for asynchronous communications and longer discussions. Navigate to http://lists.openstack.org/cgi-bin/mailman/listinfo/openstack-discuss to subscribe or view the archives.

When sending a message please prefix the *Subject* line with `[ironic]` so that we dont miss it.

Reporting Bugs

LaunchPad

Most of the tools used for OpenStack require a Launchpad ID for authentication. Ironic previously used to track work on Launchpad, but we have not done so since migrating to Storyboard.

Storyboard

The ironic project moved from Launchpad to StoryBoard for work and task tracking. This provides an aggregate view called a Project Group and individual Projects. A good starting place is the project group representing the whole of the ironic community, as opposed to the ironic project storyboard which represents ironic as a repository.

See *Bug Reporting and Triaging Guide* for more details on how we track bugs.

Contributing Code

See also:

- *So You Want to Contribute* - basic information on new code contributions
- *Developers Guide*

So You Want to Contribute

This document provides some necessary points for developers to consider when writing and reviewing Ironic code. The checklist will help developers get things right. Please make sure to check the *community page* first.

Contributing Code

If youre completely new to OpenStack and want to contribute to the ironic project, please start by familiarizing yourself with the Infra Teams Developer Guide. This will help you get your accounts set up in Launchpad and Gerrit, familiarize you with the workflow for the OpenStack continuous integration and testing systems, and help you with your first commit.

Everything Ironic

Ironic is a community of projects centered around the primary project repository ironic, which help facilitate the deployment and management of bare metal resources.

This means there are a number of different repositories that fall into the responsibility of the project team and the community. Some of the repositories may not seem strictly hardware related, but they may be tools or things to just make an aspect easier.

Related Projects

There are several projects that are tightly integrated with ironic and which are developed by the same community.

See also:

- Bifrost Documentation
- Ironic Inspector Documentation
- Ironic Lib Documentation
- Ironic Python Agent (IPA) Documentation
- Ironic Client Documentation
- Ironic Inspector Client Documentation

Adding New Features

Ironic tracks new features using RFEs (Requests for Feature Enhancements) instead of blueprints. These are stories with rfe tag, and they should be submitted before a spec or code is proposed.

When a member of the ironic-core team decides that the proposal is worth implementing, a spec (if needed) and code should be submitted, referencing the RFE task or story ID number. Contributors are welcome to submit a spec and/or code before the RFE is approved, however those patches will not land until the RFE is approved.

Feature Submission Process

1. Submit a bug report on the ironic StoryBoard. There are two fields that must be filled: Title and Description. Tasks can be added and are associated with a project. If you cant describe it in a sentence or two, it may mean that you are either trying to capture more than one RFE at once, or that you are having a hard time defining what you are trying to solve at all. This may also be a sign that your feature may require a specification document.
2. Describe the proposed change in the Description field. The description should provide enough details for a knowledgeable developer to understand what is the existing problem in the current platform that needs to be addressed, or what is the enhancement that would make the platform more capable, both from a functional and a non-functional standpoint.
3. Submit the story, add an rfe tag to it and assign yourself or whoever is going to work on this feature.

4. As soon as a member of the team acknowledges the story, we will move the story to the Review state. As time goes on, Discussion about the RFE, and whether to approve it will occur.
5. Contributors will evaluate the RFE and may advise the submitter to file a spec in the ironic-specs repository to elaborate on the feature request. Typically this is when an RFE requires extra scrutiny, more design discussion, etc. For the spec submission process, please see the *Ironic Specs Process*. A specific task should be created to track the creation of a specification.
6. If a spec is not required, once the discussion has happened and there is positive consensus among the ironic-core team on the RFE, the RFE is approved, and its tag will move from rfe to rfe-approved. This means that the feature is approved and the related code may be merged.
7. If a spec is required, the spec must be submitted (with a new task as part of the story referenced as Task in the commit message), reviewed, and merged before the RFE will be approved (and the tag changed to rfe-approved).
8. The tasks then goes through the usual process first to Review when the spec/code is being worked on, then Merged when it is implemented.
9. If the RFE is rejected, the ironic-core team will move the story to Invalid status.

Change Tracking

We track our stories and tasks in Storyboard.

https://storyboard.openstack.org/#!/project/ironic

When working on an RFE, please be sure to tag your commits properly: Story: #xxxx or Task: #xxxx. It is also helpful to set a consistent review topic, such as story/xxxx for all patches related to the RFE.

If the RFE spans across several projects (e.g. ironic and python-ironicclient), but the main work is going to happen within ironic, please use the same story for all the code youre submitting, there is no need to create a separate RFE in every project.

Note: RFEs may only be approved by members of the ironic-core team.

Note: While not strictly required for minor changes and fixes, it is highly preferred by the Ironic community that any change which needs to be backported, have a recorded Story and Task in Storyboard.

Managing Change Sets

If you would like some help, or if you (or some members of your team) are unable to continue working on the feature, updating and maintaining the changes, please let the rest of the ironic community know. You could leave a comment in one or more of the changes/patches, bring it up in IRC, the weekly meeting, or on the OpenStack development email list. Communicating this will make other contributors aware of the situation and allow for others to step forward and volunteer to continue with the work.

In the event that a contributor leaves the community, do not expect the contributors changes to be continued unless someone volunteers to do so.

Getting Your Patch Merged

Within the Ironic project, we generally require two core reviewers to sign-off (+2) change sets. We also will generally recognize non-core (+1) reviewers, and sometimes even reverse our decision to merge code based upon their reviews.

We recognize that some repositories have less visibility, as such it is okay to ask for a review in our IRC channel. Please be prepared to stay in IRC for a little while in case we have questions.

Sometimes we may also approve patches with a single core reviewer. This is generally discouraged, but sometimes necessary. When we do so, we try to explain why we do so. As a patch submitter, it equally helps us to understand why the change is important. Generally, more detail and context helps us understand the change faster.

Timeline Expectations

As with any large project, it does take time for features and changes to be merged in any of the project repositories. This is largely due to limited review bandwidth coupled with varying reviewer priorities and focuses.

When establishing an understanding of complexity, the following things should be kept in mind.

- Generally, small and minor changes can gain consensus and merge fairly quickly. These sorts of changes would be: bug fixes, minor documentation updates, follow-up changes.
- Medium changes generally consist of driver feature parity changes, where one driver is working to match functionality of another driver.
  - These changes generally only require an RFE for the purposes of tracking and correlating the change.
  - Documentation updates are expected to be submitted with or immediately following the initial change set.
- Larger or controversial changes generally take much longer to merge. This is often due to the necessity of reviewers to gain additional context and for change sets to be iterated upon to reach a state where there is consensus. These sorts of changes include: database, object, internal interface additions, RPC, rest API changes.
  - These changes will very often require specifications to reach consensus, unless there are pre-existing patterns or code already present.
  - These changes may require many reviews and iterations, and can also expect to be impacted by merge conflicts as other code or features are merged.
  - These changes must typically be split into a series of changes. Reviewers typically shy away from larger single change sets due to increased difficulty in reviewing.
  - Do not expect any API or user-visible data model changes to merge after the API client freeze. Some substrate changes may merge if not user visible.
- You should expect complex features, such as cross-project features or integration, to take longer than a single development cycle to land.
  - Building consensus is vital.

  - Often these changes are controversial or have multiple considerations that need to be worked through in the specification process, which may cause the design to change. As such, it may take months to reach consensus over design.
  - These features are best broken into larger chunks and tackled in an incremental fashion.

Live Upgrade Related Concerns

See *Rolling Upgrades*.

Driver Internal Info

The `driver_internal_info` node field was introduced in the Kilo release. It allows driver developers to store internal information that can not be modified by end users. Here is the list of existing common and agent driver attributes:

- Common attributes:
  - `is_whole_disk_image`: A Boolean value to indicate whether the user image contains ramdisk/kernel.
  - `clean_steps`: An ordered list of clean steps that will be performed on the node.
  - `deploy_steps`: An ordered list of deploy steps that will be performed on the node. Support for deploy steps was added in the `11.1.0` release.
  - `instance`: A list of dictionaries containing the disk layout values.
  - `root_uuid_or_disk_id`: A String value of the bare metal nodes root partition uuid or disk id.
  - `persistent_boot_device`: A String value of device from `ironic.common.boot_devices`.
  - `is_next_boot_persistent`: A Boolean value to indicate whether the next boot device is `persistent_boot_device`.
- Agent driver attributes:
  - `agent_url`: A String value of IPA API URL so that Ironic can talk to IPA ramdisk.
  - `hardware_manager_version`: A String value of the version of the hardware manager in IPA ramdisk.
  - `target_raid_config`: A Dictionary containing the target RAID configuration. This is a copy of the same name attribute in Node object. But this one is never actually saved into DB and is only read by IPA ramdisk.

Note: These are only some fields in use. Other vendor drivers might expose more `driver_internal_info` properties, please check their development documentation and/or module docstring for details. It is important for developers to make sure these properties follow the precedent of prefixing their variable names with a specific interface name (e.g., ilo_bar, drac_xyz), so as to minimize or avoid any conflicts between interfaces.

Ironic Specs Process

Specifications must follow the template which can be found at specs/template.rst, which is quite self-documenting. Specifications are proposed by adding them to the *specs/approved* directory, adding a soft link to it from the *specs/not-implemented* directory, and posting it for review to Gerrit. For more information, please see the README.

The same Gerrit process as with source code, using the repository ironic-specs, is used to add new specifications.

All approved specifications are available at: https://specs.openstack.org/openstack/ironic-specs. If a specification has been approved but not completed within one or more releases since the approval, it may be re-reviewed to make sure it still makes sense as written.

Ironic specifications are part of the *RFE (Requests for Feature Enhancements) process*. You are welcome to submit patches associated with an RFE, but they will have a -2 (do not merge) until the specification has been approved. This is to ensure that the patches dont get accidentally merged beforehand. You will still be able to get reviewer feedback and push new patch sets, even with a -2. The list of core reviewers for the specifications is small but mighty. (This is not necessarily the same list of core reviewers for code patches.)

Changes to existing specs

For approved but not-completed specs:

- cosmetic cleanup, fixing errors, and changing the definition of a feature can be done to the spec.

For approved and completed specs:

- changing a previously approved and completed spec should only be done for cosmetic cleanup or fixing errors.
- changing the definition of the feature should be done in a new spec.

Please see the Ironic specs process wiki page for further reference.

Project Team Leader Duties

The `Project Team Leader` or `PTL` is elected each development cycle by the contributors to the ironic community.

Think of this person as your primary contact if you need to try and rally the project, or have a major issue that requires attention.

They serve a role that is mainly oriented towards trying to drive the technical discussion forward and managing the idiosyncrasies of the project. With this responsibility, they are considered a public face of the project and are generally obliged to try and provide project updates and outreach communication.

All common PTL duties are enumerated here in the PTL guide.

Tasks like release management or preparation for a release are generally delegated with-in the team. Even outreach can be delegated, and specifically there is no rule stating that any member of the community cant propose a release, clean-up release notes or documentation, or even get on the occasional stage.

Bug Reporting and Triaging Guide

StoryBoard

All ironic projects use StoryBoard for tracking both bugs and enhancement requests (RFE). The ironic project group lists all our projects.

Note: Ironic is developed as part of OpenStack and therefore uses the `openstack/` namespace.

StoryBoard is somewhat different from traditional bug tracking systems because every *story* is not linked to a project itself, but rather through its *tasks*. A story represents an issue you are facing or an enhancement you want to see, while tasks represent individual action items which can span several projects. When creating a story, youll also need to create the first task. If unsure, create a task against `openstack/ironic`.

Reporting Guide

We are constantly receiving a lot of requests, so its important to file a meaningful story for it to be acted upon. A good story:

- specifies **why** a change is needed. In case of a bug - what you expected to happen.
- explains how to reproduce the described condition.

  Note: Please try to provide a reproducer based on unit tests, *devstack* or bifrost. While we try our best to support users using other installers and distributions, it may be non-trivial without deep knowledge of them. If youre using a commercial distribution or a product, please try contacting support first.

- should be understandable without additional context. For example, if you see an exception, we will need the full traceback. Other commonly required things are:
  - the contents of the node in question (use `baremetal node show <uuid>`)
  - debug logging related to the event, ideally with logs from the ramdisk
  - versions of ironic, ironic-python-agent, and any other coupled components.
- should not be too verbose either. Unfortunately, we cannot process a few days worth of system logs to find the problems, we expect your collaboration.
- is not a question or a support request. Please see *So You Want to Contribute* for the ways to contact us.
- provides a way to contact the reporter. Please follow the comments and expect follow-up emails, but ideally also be on IRC for questions.

An enhancement request additionally:

- benefits the overall project, not just one consumer. If you have a case that is specific to your requirements, think about ways to make ironic extensible to be able to cover it.
- does not unnecessary increase the project scope. Consider if your idea can be implemented without changing ironic or its projects, maybe it actually should?

Triaging Guide

The bug triaging process involves checking new stories to make sure they are actionable by the team. This guide is mostly targeting the project team, but we would appreciate if reporters could partly self-triage their own requests.

- Determine if the request is valid and complete. Use the checklist in the *Reporting Guide* for that.
- Is the request a bug report or an enhancement request (an RFE)? The difference is often subtle, the key question to answer is if the described behavior is expected.

  Add an `rfe` tag to all enhancement requests and propose it for the RFE Review section of the weekly meeting.
- Does the RFE obviously require a spec? Usually this is decided when an RFE is reviewed during the meeting, but some requests are undoubtedly complex, involve changing a lot of critical parts and thus demand a spec.

  Add a `needs-spec` tag to enhancement requests that obviously need a spec. Otherwise leave it until the meeting.
- Apply additional tags:
  - All hardware type specific stories should receive a corresponding tag (e.g. `ipmi`, `idrac`, etc).
  - API-related stories should have an `api` tag.
  - CI issues should have a `gate` tag.

The next actions **must only** be done by a core team member (or an experienced full-time contributor appoined by the PTL):

- Can the RFE be automatically approved? It happens if the RFE requests an implementation of a driver feature that is already implemented for other drivers and does not pose additional complexity.

  If the RFE can be automatically approved, apply the `rfe-approved` tag. If unsure, never apply the tag! Talk to the PTL instead.
- Does the RFE have a corresponding spec approved? If yes, apply the `rfe-approved` tag.
- In the end, apply the `ironic-triaged` tag to make the story as triaged.

Expiring Bugs

While we hope to fix all issues that our consumers hit, it is unfortunately not realistic. Stories **may** be closed by marking all their tasks `INVALID` in the following cases:

- No solution has been proposed in 1 calendar year.
- Additional information has been requested from the reporter, and no update has been provided in 1 calendar month.
- The request no longer aligns with the direction of the project.

Note: As usual, common sense should be applied when closing stories.

Developer Quick-Start

This is a quick walkthrough to get you started developing code for Ironic. This assumes you are already familiar with submitting code reviews to an OpenStack project.

The gate currently runs the unit tests under Python 3.6, 3.7, and 3.8. It is strongly encouraged to run the unit tests locally prior to submitting a patch.

Note: Do not run unit tests on the same environment as devstack due to conflicting configuration with system dependencies.

Note: This document is compatible with Python (3.6, 3.7, 3.8), Debian buster (10.8), Ubuntu Focal Fossa (20.04 LTS), RHEL8/CentOS Stream, openSUSE/SLE 15, and Fedora (33). When referring to different versions of Python and OS distributions, this is explicitly stated.

See also:

https://docs.openstack.org/infra/manual/developers.html#development-workflow

Prepare Development System

System Prerequisites

The following packages cover the prerequisites for a local development environment on most current distributions. Instructions for getting set up with non-default versions of Python and on older distributions are included below as well.

- Ubuntu/Debian:

```
sudo apt-get install build-essential python3-dev libssl-dev python3-pip␣
↪libmysqlclient-dev libxml2-dev libxslt-dev libpq-dev git git-review␣
↪libffi-dev gettext ipmitool psmisc graphviz libjpeg-dev
```

- RHEL/CentOS/Fedora:

```
sudo dnf install python3-devel openssl-devel python3-pip mysql-devel␣
↪libxml2-devel libxslt-devel postgresql-devel git git-review libffi-
↪devel gettext ipmitool psmisc graphviz gcc libjpeg-turbo-devel
```

- openSUSE/SLE:

```
sudo zypper install git git-review libffi-devel libmysqlclient-devel␣
↪libopenssl-devel libxml2-devel libxslt-devel postgresql-devel python3-
↪devel python-nose python3-pip gettext-runtime psmisc
```

To run the tests locally, it is a requirement that your terminal emulator supports unicode with the `en_US.UTF8` locale. If you use locale-gen to manage your locales, make sure you have enabled `en_US.UTF8` in `/etc/locale.gen` and rerun `locale-gen`.

Python Prerequisites

We suggest to use at least tox 3.9, if your distribution has an older version, you can install it using pip system-wise or better per user using the user option that by default will install the binary under $HOME/.local/bin, so you need to be sure to have that path in $PATH; for example:

```
pip install tox --user
```

will install tox as ~/.local/bin/tox

You may need to explicitly upgrade virtualenv if youve installed the one from your OS distribution and it is too old (tox will complain). You can upgrade it individually, if you need to:

```
pip install -U virtualenv --user
```

Running Unit Tests Locally

If you havent already, Ironic source code should be pulled directly from git:

```
# from your home or source directory
cd ~
git clone https://opendev.org/openstack/ironic
cd ironic
```

Running Unit and Style Tests

All unit tests should be run using tox. To run Ironics entire test suite:

```
# to run the py3 unit tests, and the style tests
tox
```

To run a specific test or tests, use the -e option followed by the tox target name. For example:

```
# run the unit tests under py36 and also run the pep8 tests
tox -epy36 -epep8
```

You may pass options to the test programs using positional arguments. To run a specific unit test, this passes the desired test (regex string) to stestr:

```
# run a specific test for Python 3.6
tox -epy36 -- test_conductor
```

Debugging unit tests

In order to break into the debugger from a unit test we need to insert a breaking point to the code:

```
import pdb; pdb.set_trace()
```

Then run `tox` with the debug environment as one of the following:

```
tox -e debug
tox -e debug test_file_name
tox -e debug test_file_name.TestClass
tox -e debug test_file_name.TestClass.test_name
```

For more information see the oslotest documentation.

Database Setup

The unit tests need a local database setup, you can use `tools/test-setup.sh` to set up the database the same way as setup in the OpenStack test systems.

Additional Tox Targets

There are several additional tox targets not included in the default list, such as the target which builds the documentation site. See the `tox.ini` file for a complete listing of tox targets. These can be run directly by specifying the target name:

```
# generate the documentation pages locally
tox -edocs

# generate the sample configuration file
tox -egenconfig
```

Exercising the Services Locally

In addition to running automated tests, sometimes it can be helpful to actually run the services locally, without needing a server in a remote datacenter.

If you would like to exercise the Ironic services in isolation within your local environment, you can do this without starting any other OpenStack services. For example, this is useful for rapidly prototyping and debugging interactions over the RPC channel, testing database migrations, and so forth.

Here we describe two ways to install and configure the dependencies, either run directly on your local machine or encapsulated in a virtual machine or container.

Step 1: Create a Python virtualenv

1. If you havent already downloaded the source code, do that first:

```
cd ~
git clone https://opendev.org/openstack/ironic
cd ironic
```

2. Create the Python virtualenv:

```
tox -evenv --notest --develop -r
```

3. Activate the virtual environment:

```
. .tox/venv/bin/activate
```

4. Install the *openstack* client command utility:

```
pip install python-openstackclient
```

5. Install the *baremetal* client:

```
pip install python-ironicclient
```

> **Note:** You can install python-ironicclient from source by cloning the git repository and running *pip install .* while in the root of the cloned repository.

6. Export some ENV vars so the client will connect to the local services that youll start in the next section:

```
export OS_AUTH_TYPE=none
export OS_ENDPOINT=http://localhost:6385/
```

Next, install and configure system dependencies.

Step 2: Install System Dependencies Locally

This step will install MySQL on your local system. This may not be desirable in some situations (eg, youre developing from a laptop and do not want to run a MySQL server on it all the time). If you want to use SQLite, skip it and do not set the `connection` option.

1. Install mysql-server:

   Ubuntu/Debian:

```
sudo apt-get install mysql-server
```

   RHEL/CentOS/Fedora:

```
sudo dnf install mariadb mariadb-server
sudo systemctl start mariadb.service
```

openSUSE/SLE:: sudo zypper install mariadb sudo systemctl start mysql.service

If using MySQL, you need to create the initial database:

```
mysql -u root -pMYSQL_ROOT_PWD -e "create schema ironic"
```

Note: if you choose not to install mysql-server, ironic will default to using a local sqlite database. The database will then be stored in `ironic/ironic.sqlite`.

2. Create a configuration file within the ironic source directory:

```
# generate a sample config
tox -egenconfig

# copy sample config and modify it as necessary
cp etc/ironic/ironic.conf.sample etc/ironic/ironic.conf.local

# disable auth since we are not running keystone here
sed -i "s/#auth_strategy = keystone/auth_strategy = noauth/" etc/ironic/
↪ironic.conf.local

# use the 'fake-hardware' test hardware type
sed -i "s/#enabled_hardware_types = .*/enabled_hardware_types = fake-
↪hardware/" etc/ironic/ironic.conf.local

# use the 'fake' deploy and boot interfaces
sed -i "s/#enabled_deploy_interfaces = .*/enabled_deploy_interfaces =␣
↪fake/" etc/ironic/ironic.conf.local
sed -i "s/#enabled_boot_interfaces = .*/enabled_boot_interfaces = fake/"␣
↪etc/ironic/ironic.conf.local

# enable both fake and ipmitool management and power interfaces
sed -i "s/#enabled_management_interfaces = .*/enabled_management_
↪interfaces = fake,ipmitool/" etc/ironic/ironic.conf.local
sed -i "s/#enabled_power_interfaces = .*/enabled_power_interfaces = fake,
↪ipmitool/" etc/ironic/ironic.conf.local

# change the periodic sync_power_state_interval to a week, to avoid␣
↪getting NodeLocked exceptions
sed -i "s/#sync_power_state_interval = 60/sync_power_state_interval =␣
↪604800/" etc/ironic/ironic.conf.local

# if you opted to install mysql-server, switch the DB connection from␣
↪sqlite to mysql
sed -i "s/#connection = .*/connection = mysql\+pymysql:\/\/root:MYSQL_
↪ROOT_PWD@localhost\/ironic/" etc/ironic/ironic.conf.local

# use JSON RPC to avoid installing rabbitmq locally
sed -i "s/#rpc_transport = oslo/rpc_transport = json-rpc/" etc/ironic/
↪ironic.conf.local
```

Step 3: Start the Services

From within the python virtualenv, run the following command to prepare the database before you start the ironic services:

```
# initialize the database for ironic
ironic-dbsync --config-file etc/ironic/ironic.conf.local create_schema
```

Next, open two new terminals for this section, and run each of the examples here in a separate terminal. In this way, the services will *not* be run as daemons; you can observe their output and stop them with Ctrl-C at any time.

1. Start the API service in debug mode and watch its output:

   ```
   cd ~/ironic
   . .tox/venv/bin/activate
   ironic-api -d --config-file etc/ironic/ironic.conf.local
   ```

2. Start the Conductor service in debug mode and watch its output:

   ```
   cd ~/ironic
   . .tox/venv/bin/activate
   ironic-conductor -d --config-file etc/ironic/ironic.conf.local
   ```

Step 4: Interact with the running services

You should now be able to interact with ironic via the python client, which is present in the python virtualenv, and observe both services debug outputs in the other two windows. This is a good way to test new features or play with the functionality without necessarily starting DevStack.

To get started, export the following variables to point the client at the local instance of ironic and disable the authentication:

```
export OS_AUTH_TYPE=none
export OS_ENDPOINT=http://127.0.0.1:6385
```

Then list the available commands and resources:

```
# get a list of available commands
openstack help baremetal

# get the list of drivers currently supported by the available conductor(s)
baremetal driver list

# get a list of nodes (should be empty at this point)
baremetal node list
```

Here is an example walkthrough of creating a node:

```
MAC="aa:bb:cc:dd:ee:ff"   # replace with the MAC of a data port on your node
IPMI_ADDR="1.2.3.4"       # replace with a real IP of the node BMC
```

(continues on next page)

(continued from previous page)

```
IPMI_USER="admin"         # replace with the BMC's user name
IPMI_PASS="pass"          # replace with the BMC's password

# enroll the node with the fake hardware type and IPMI-based power and
# management interfaces. Note that driver info may be added at node
# creation time with "--driver-info"
NODE=$(baremetal node create \
       --driver fake-hardware \
       --management-interface ipmitool \
       --power-interface ipmitool \
       --driver-info ipmi_address=$IPMI_ADDR \
       --driver-info ipmi_username=$IPMI_USER \
       -f value -c uuid)

# driver info may also be added or updated later on
baremetal node set $NODE --driver-info ipmi_password=$IPMI_PASS

# add a network port
baremetal port create $MAC --node $NODE

# view the information for the node
baremetal node show $NODE

# request that the node's driver validate the supplied information
baremetal node validate $NODE

# you have now enrolled a node sufficiently to be able to control
# its power state from ironic!
baremetal node power on $NODE
```

If you make some code changes and want to test their effects, simply stop the services with Ctrl-C and restart them.

Step 5: Fixing your test environment

If you are testing changes that add or remove python entrypoints, or making significant changes to ironics python modules, or simply keep the virtualenv around for a long time, your development environment may reach an inconsistent state. It may help to delete cached .pyc files, update dependencies, reinstall ironic, or even recreate the virtualenv. The following commands may help with that, but are not an exhaustive troubleshooting guide:

```
# clear cached pyc files
cd ~/ironic/ironic
find ./ -name '*.pyc' | xargs rm

# reinstall ironic modules
cd ~/ironic
. .tox/venv/bin/activate
pip uninstall ironic
```

(continues on next page)

(continued from previous page)

```
pip install -e .

# install and upgrade ironic and all python dependencies
cd ~/ironic
. .tox/venv/bin/activate
pip install -U -e .
```

Deploying Ironic with DevStack

DevStack may be configured to deploy Ironic, setup Nova to use the Ironic driver and provide hardware resources (network, baremetal compute nodes) using a combination of OpenVSwitch and libvirt. It is highly recommended to deploy on an expendable virtual machine and not on your personal work station. Deploying Ironic with DevStack requires a machine running Ubuntu 16.04 (or later) or Fedora 24 (or later). Make sure your machine is fully up to date and has the latest packages installed before beginning this process.

The ironic-tempest-plugin is necessary if you want to run integration tests, the section *Ironic with ironic-tempest-plugin* tells the extra steps you need to enable it in DevStack.

See also:

https://docs.openstack.org/devstack/latest/

Note: The devstack demo tenant is now granted the baremetal_observer role and thereby has read-only access to ironics API. This is sufficient for all the examples below. Should you want to create or modify bare metal resources directly (ie. through ironic rather than through nova) you will need to use the devstack admin tenant.

Devstack will no longer create the user stack with the desired permissions, but does provide a script to perform the task:

```
git clone https://opendev.org/openstack/devstack.git devstack
sudo ./devstack/tools/create-stack-user.sh
```

Note: In case you receive an error Could not determine host ip address. See local.conf for suggestions on setting HOST_IP, you need to manually add the main ip of your machine to the localrc file under devstack/ using the HOST_IP variable, e.g. HOST_IP=YOURIP This could happen when running devstack on virtual machines.

Switch to the stack user and clone DevStack:

```
sudo su - stack
git clone https://opendev.org/openstack/devstack.git devstack
```

Ironic

Create devstack/local.conf with minimal settings required to enable Ironic. An example local.conf that enables the `direct` *deploy interface* and uses the `ipmi` hardware type by default:

```
cd devstack
cat >local.conf <<END
[[local|localrc]]
# Enable only minimal services
disable_all_services
enable_service g-api
enable_service key
enable_service memory_tracker
enable_service mysql
enable_service q-agt
enable_service q-dhcp
enable_service q-l3
enable_service q-meta
enable_service q-metering
enable_service q-svc
enable_service rabbit

# Credentials
ADMIN_PASSWORD=password
DATABASE_PASSWORD=password
RABBIT_PASSWORD=password
SERVICE_PASSWORD=password
SERVICE_TOKEN=password

# Set glance's default limit to be baremetal image friendly
GLANCE_LIMIT_IMAGE_SIZE_TOTAL=5000

# Enable Ironic plugin
enable_plugin ironic https://opendev.org/openstack/ironic

# Create 3 virtual machines to pose as Ironic's baremetal nodes.
IRONIC_VM_COUNT=3
IRONIC_BAREMETAL_BASIC_OPS=True
DEFAULT_INSTANCE_TYPE=baremetal

IRONIC_RPC_TRANSPORT=json-rpc
IRONIC_RAMDISK_TYPE=tinyipa

# Enable additional hardware types, if needed.
#IRONIC_ENABLED_HARDWARE_TYPES=ipmi,fake-hardware
# Don't forget that many hardware types require enabling of additional
# interfaces, most often power and management:
#IRONIC_ENABLED_MANAGEMENT_INTERFACES=ipmitool,fake
#IRONIC_ENABLED_POWER_INTERFACES=ipmitool,fake
#IRONIC_DEFAULT_DEPLOY_INTERFACE=direct
```

(continues on next page)

(continued from previous page)

```
# Change this to alter the default driver for nodes created by devstack.
# This driver should be in the enabled list above.
IRONIC_DEPLOY_DRIVER="ipmi"

# The parameters below represent the minimum possible values to create
# functional nodes.
IRONIC_VM_SPECS_RAM=1024
IRONIC_VM_SPECS_DISK=3

# Size of the ephemeral partition in GB. Use 0 for no ephemeral partition.
IRONIC_VM_EPHEMERAL_DISK=0

# To build your own IPA ramdisk from source, set this to True
IRONIC_BUILD_DEPLOY_RAMDISK=False

INSTALL_TEMPEST=False
VIRT_DRIVER=ironic

# By default, DevStack creates a 10.0.0.0/24 network for instances.
# If this overlaps with the hosts network, you may adjust with the
# following.
IP_VERSION=4
FIXED_RANGE=10.1.0.0/20
IPV4_ADDRS_SAFE_TO_USE=10.1.0.0/20
NETWORK_GATEWAY=10.1.0.1

Q_AGENT=openvswitch
Q_ML2_PLUGIN_MECHANISM_DRIVERS=openvswitch
Q_ML2_TENANT_NETWORK_TYPE=vxlan

# Log all output to files
LOGFILE=/opt/stack/devstack.log
LOGDIR=/opt/stack/logs
IRONIC_VM_LOG_DIR=/opt/stack/ironic-bm-logs

END
```

Ironic with ironic-tempest-plugin

Using the stack user, clone the ironic-tempest-plugin repository in the same directory you cloned DevStack:

```
git clone https://opendev.org/openstack/ironic-tempest-plugin.git
```

An example local.conf that enables the ironic tempest plugin and Ironic can be found below. The `TEMPEST_PLUGINS` variable needs to have the absolute path to the ironic-tempest-plugin folder, otherwise the plugin wont be installed. Ironic will have enabled the `direct` *deploy interface* and uses the `ipmi` hardware type by default:

```
cd devstack
cat >local.conf <<END
[[local|localrc]]
# Credentials
ADMIN_PASSWORD=password
DATABASE_PASSWORD=password
RABBIT_PASSWORD=password
SERVICE_PASSWORD=password
SERVICE_TOKEN=password
SWIFT_HASH=password
SWIFT_TEMPURL_KEY=password

# Set glance's default limit to be baremetal image friendly
GLANCE_LIMIT_IMAGE_SIZE_TOTAL=5000

# Enable Ironic plugin
enable_plugin ironic https://opendev.org/openstack/ironic

# Disable nova novnc service, ironic does not support it anyway.
disable_service n-novnc

# Enable Swift for the direct deploy interface.
enable_service s-proxy
enable_service s-object
enable_service s-container
enable_service s-account

# Disable Horizon
disable_service horizon

# Disable Cinder
disable_service cinder c-sch c-api c-vol

# Swift temp URL's are required for the direct deploy interface
SWIFT_ENABLE_TEMPURLS=True

# Create 3 virtual machines to pose as Ironic's baremetal nodes.
IRONIC_VM_COUNT=3
IRONIC_BAREMETAL_BASIC_OPS=True
DEFAULT_INSTANCE_TYPE=baremetal

# Enable additional hardware types, if needed.
#IRONIC_ENABLED_HARDWARE_TYPES=ipmi,fake-hardware
# Don't forget that many hardware types require enabling of additional
# interfaces, most often power and management:
#IRONIC_ENABLED_MANAGEMENT_INTERFACES=ipmitool,fake
#IRONIC_ENABLED_POWER_INTERFACES=ipmitool,fake
#IRONIC_DEFAULT_DEPLOY_INTERFACE=direct

# Change this to alter the default driver for nodes created by devstack.
```

(continues on next page)

(continued from previous page)

```
# This driver should be in the enabled list above.
IRONIC_DEPLOY_DRIVER=ipmi

# The parameters below represent the minimum possible values to create
# functional nodes.
IRONIC_VM_SPECS_RAM=2048
IRONIC_VM_SPECS_DISK=10

# Size of the ephemeral partition in GB. Use 0 for no ephemeral partition.
IRONIC_VM_EPHEMERAL_DISK=0

# To build your own IPA ramdisk from source, set this to True
IRONIC_BUILD_DEPLOY_RAMDISK=False

VIRT_DRIVER=ironic

# By default, DevStack creates a 10.0.0.0/24 network for instances.
# If this overlaps with the hosts network, you may adjust with the
# following.
NETWORK_GATEWAY=10.1.0.1
FIXED_RANGE=10.1.0.0/24
FIXED_NETWORK_SIZE=256

# Log all output to files
LOGFILE=$HOME/devstack.log
LOGDIR=$HOME/logs
IRONIC_VM_LOG_DIR=$HOME/ironic-bm-logs
TEMPEST_PLUGINS="/opt/stack/ironic-tempest-plugin"

END
```

Note: Some tests may be skipped depending on the configuration of your environment, they may be reliant on a driver or a capability that you did not configure.

Deployment

Note: Git protocol requires access to port 9418, which is not a standard port that corporate firewalls always allow. If you are behind a firewall or on a proxy that blocks Git protocol, modify the `enable_plugin` line to use `https://` instead of `git://` and add `GIT_BASE=https://opendev.org` to the credentials:

```
GIT_BASE=https://opendev.org

# Enable Ironic plugin
enable_plugin ironic https://opendev.org/openstack/ironic
```

Note: When the `ipmi` hardware type is used and IRONIC_IS_HARDWARE variable is `false` devstack will automatically set up VirtualBMC to control the power state of the virtual baremetal nodes.

Note: When running QEMU as non-root user (e.g. `qemu` on Fedora or `libvirt-qemu` on Ubuntu), make sure `IRONIC_VM_LOG_DIR` points to a directory where QEMU will be able to write. You can verify this with, for example:

```
# on Fedora
sudo -u qemu touch $HOME/ironic-bm-logs/test.log
# on Ubuntu
sudo -u libvirt-qemu touch $HOME/ironic-bm-logs/test.log
```

Note: To check out an in-progress patch for testing, you can add a Git ref to the `enable_plugin` line. For instance:

```
enable_plugin ironic https://opendev.org/openstack/ironic refs/changes/46/
↪295946/15
```

For a patch in review, you can find the ref to use by clicking the Download button in Gerrit. You can also specify a different git repo, or a branch or tag:

```
enable_plugin ironic https://github.com/openstack/ironic stable/kilo
```

For more details, see the devstack plugin interface documentation.

Run stack.sh:

```
./stack.sh
```

Source credentials, create a key, and spawn an instance as the `demo` user:

```
. ~/devstack/openrc

# query the image id of the default cirros image
image=$(openstack image show $DEFAULT_IMAGE_NAME -f value -c id)

# create keypair
ssh-keygen
openstack keypair create --public-key ~/.ssh/id_rsa.pub default

# spawn instance
openstack server create --flavor baremetal --image $image --key-name default␣
↪testing
```

Note: Because devstack create multiple networks, we need to pass an additional parameter `--nic net-id` to the nova boot command when using the admin account, for example:

```
net_id=$(openstack network list | egrep "$PRIVATE_NETWORK_NAME"'[^-]' | awk '
↪{ print $2 }')

openstack server create --flavor baremetal --nic net-id=$net_id --image
↪$image --key-name default testing
```

You should now see a Nova instance building:

```
openstack server list --long
+----------+---------+--------+------------+-------------+----------+---------
↪---+----------+------------------+------+------------+
| ID       | Name    | Status | Task State | Power State | Networks | Image␣
↪Name | Image ID | Availability Zone | Host | Properties |
+----------+---------+--------+------------+-------------+----------+---------
↪---+----------+------------------+------+------------+
| a2c7f812 | testing | BUILD  | spawning   | NOSTATE     |          | cirros-
↪0.3 | 44d4092a | nova              |      |            |
| -e386-4a |         |        |            |             |          | .5-x86_
↪64- | -51ac-47 |                   |      |            |
| 22-b393- |         |        |            |             |          | disk   ␣
↪    | 51-9c50- |                   |      |            |
| fe1802ab |         |        |            |             |          |        ␣
↪    | fd6e2050 |                   |      |            |
| d56e     |         |        |            |             |          |        ␣
↪    | faa1     |                   |      |            |
+----------+---------+--------+------------+-------------+----------+---------
↪---+----------+------------------+------+------------+
```

Nova will be interfacing with Ironic conductor to spawn the node. On the Ironic side, you should see an Ironic node associated with this Nova instance. It should be powered on and in a wait call-back provisioning state:

```
baremetal node list
+--------------------------------------+--------+--------------------------------
↪--------+-------------+--------------------+-------------+
| UUID                                 | Name   | Instance UUID                 ␣
↪         | Power State | Provisioning State | Maintenance |
+--------------------------------------+--------+--------------------------------
↪--------+-------------+--------------------+-------------+
| 9e592cbe-e492-4e4f-bf8f-4c9e0ad1868f | node-0 | None                          ␣
↪         | power off   | None               | False       |
| ec0c6384-cc3a-4edf-b7db-abde1998be96 | node-1 | None                          ␣
↪         | power off   | None               | False       |
| 4099e31c-576c-48f8-b460-75e1b14e497f | node-2 | a2c7f812-e386-4a22-b393-
↪fe1802abd56e | power on    | wait call-back     | False       |
+--------------------------------------+--------+--------------------------------
↪--------+-------------+--------------------+-------------+
```

At this point, Ironic conductor has called to libvirt (via virtualbmc) to power on a virtual machine, which will PXE + TFTP boot from the conductor node and progress through the Ironic provisioning workflow.

One libvirt domain should be active now:

```
sudo virsh list --all
 Id    Name                           State
----------------------------------------------------
 2     node-2                         running
 -     node-0                         shut off
 -     node-1                         shut off
```

This provisioning process may take some time depending on the performance of the host system, but Ironic should eventually show the node as having an active provisioning state:

```
baremetal node list
+--------------------------------------+--------+--------------------------------
↪--------+-------------+-------------------+-------------+
| UUID                                 | Name   | Instance UUID                  ␣
↪         | Power State | Provisioning State | Maintenance |
+--------------------------------------+--------+--------------------------------
↪--------+-------------+-------------------+-------------+
| 9e592cbe-e492-4e4f-bf8f-4c9e0ad1868f | node-0 | None                           ␣
↪         | power off   | None               | False       |
| ec0c6384-cc3a-4edf-b7db-abde1998be96 | node-1 | None                           ␣
↪         | power off   | None               | False       |
| 4099e31c-576c-48f8-b460-75e1b14e497f | node-2 | a2c7f812-e386-4a22-b393-
↪fe1802abd56e | power on    | active             | False       |
+--------------------------------------+--------+--------------------------------
↪--------+-------------+-------------------+-------------+
```

This should also be reflected in the Nova instance state, which at this point should be ACTIVE, Running and an associated private IP:

```
openstack server list --long
+----------+---------+--------+------------+-------------+---------------+----
↪--------+----------+-------------------+------+------------+
| ID       | Name    | Status | Task State | Power State | Networks      |␣
↪Image Name | Image ID | Availability Zone | Host | Properties |
+----------+---------+--------+------------+-------------+---------------+----
↪--------+----------+-------------------+------+------------+
| a2c7f812 | testing | ACTIVE | none       | Running     | private=10.1. |␣
↪cirros-0.3 | 44d4092a | nova              |      |            |
| -e386-4a |         |        |            |             | 0.4, fd7d:1f3 | .5-
↪x86_64- | -51ac-47 |                   |      |            |
| 22-b393- |         |        |            |             | c:4bf1:0:f816 |␣
↪disk       | 51-9c50- |                   |      |            |
| fe1802ab |         |        |            |             | :3eff:f39d:6d |  ␣
↪        | fd6e2050 |                   |      |            |
| d56e     |         |        |            |             | 94            |  ␣
↪        | faa1     |                   |      |            |
+----------+---------+--------+------------+-------------+---------------+----
↪--------+----------+-------------------+------+------------+
```

The server should now be accessible via SSH:

```
ssh cirros@10.1.0.4
$
```

Running Tempest tests

After *Deploying Ironic with DevStack* with the ironic-tempest-plugin enabled, one might want to run integration tests against the running cloud. The Tempest project is the project that offers an integration test suite for OpenStack.

First, navigate to Tempest directory:

```
cd /opt/stack/tempest
```

To run all tests from the Ironic plugin, execute the following command:

```
tox -e all -- ironic
```

To limit the amount of tests that you would like to run, you can use a regex. For instance, to limit the run to a single test file, the following command can be used:

```
tox -e all -- ironic_tempest_plugin.tests.scenario.test_baremetal_basic_ops
```

Debugging Tempest tests

It is sometimes useful to step through the test code, line by line, especially when the error output is vague. This can be done by running the tests in debug mode and using a debugger such as pdb.

For example, after editing the *test_baremetal_basic_ops* file and setting up the pdb traces you can invoke the `run_tempest.sh` script in the Tempest directory with the following parameters:

```
./run_tempest.sh -N -d ironic_tempest_plugin.tests.scenario.test_baremetal_
↪basic_ops
```

- The *-N* parameter tells the script to run the tests in the local environment (without a virtualenv) so it can find the Ironic tempest plugin.
- The *-d* parameter enables the debug mode, allowing it to be used with pdb.

For more information about the supported parameters see:

```
./run_tempest.sh --help
```

Note: Always be careful when running debuggers in time sensitive code, they may cause timeout errors that werent there before.

OSProfiler Tracing in Ironic

OSProfiler is an OpenStack cross-project profiling library. It is being used among OpenStack projects to look at performance issues and detect bottlenecks. For details on how OSProfiler works and how to use it in ironic, please refer to OSProfiler Support Documentation.

Building developer documentation

If you would like to build the documentation locally, eg. to test your documentation changes before uploading them for review, run these commands to build the documentation set:

- On your local machine:

```
# activate your development virtualenv
. .tox/venv/bin/activate

# build the docs
tox -edocs

#Now use your browser to open the top-level index.html located at:

ironic/doc/build/html/index.html
```

- On a remote machine:

```
# Go to the directory that contains the docs
cd ~/ironic/doc/source/

# Build the docs
tox -edocs

# Change directory to the newly built HTML files
cd ~/ironic/doc/build/html/

# Create a server using python on port 8000
python -m SimpleHTTPServer 8000

#Now use your browser to open the top-level index.html located at:

http://your_ip:8000
```

Developer FAQ (frequently asked questions)

Here are some answers to frequently-asked questions from IRC and elsewhere.

How do I

create a migration script template?

Using the `ironic-dbsync revision` command, e.g:

```
$ cd ironic
$ tox -evenv -- ironic-dbsync revision -m \"create foo table\"
```

It will create an empty alembic migration. For more information see the alembic documentation.

know if a release note is needed for my change?

Reno documentation contains a description of what can be added to each section of a release note. If, after reading this, youre still unsure about whether to add a release note for your change or not, keep in mind that it is intended to contain information for deployers, so changes to unit tests or documentation are unlikely to require one.

create a new release note?

By running `reno` command via tox, e.g:

```
$ tox -e venv -- reno new version-foo
  venv create: /home/foo/ironic/.tox/venv
  venv installdeps: -r/home/foo/ironic/test-requirements.txt
  venv develop-inst: /home/foo/ironic
  venv runtests: PYTHONHASHSEED='0'
  venv runtests: commands[0] | reno new version-foo
  Created new notes file in releasenotes/notes/version-foo-ecb3875dc1cbf6d9.
↪yaml
    venv: commands succeeded
    congratulations :)
```

(continues on next page)

(continued from previous page)

```
$ git status
  On branch test
  Untracked files:
    (use "git add <file>..." to include in what will be committed)

    releasenotes/notes/version-foo-ecb3875dc1cbf6d9.yaml
```

Then edit the result file. Note that:

- we prefer to use present tense in release notes. For example, a release note should say Adds support for feature foo, not Added support for feature foo. (We use adds instead of add because grammatically, it is ironic adds support, not ironic add support.)
- any variant of English spelling (American, British, Canadian, Australian) is acceptable. The release note itself should be consistent and not have different spelling variants of the same word.

For more information see the reno documentation.

update a release note?

If this is a release note that pertains to something that was fixed on master or an intermediary release (during a development cycle, that hasnt been branched yet), you can go ahead and update it by submitting a patch.

If it is the release note of an ironic release that has branched, it can be updated but we will only allow it in extenuating circumstances. (It can be updated by *only* updating the file in that branch. DO NOT update the file in master and cherry-pick it. If you do, see how the mess was cleaned up.)

get a decision on something?

You have an issue and would like a decision to be made. First, make sure that the issue hasnt already been addressed, by looking at documentation, stories, specifications, or asking. Information and links can be found on the Ironic wiki page.

There are several ways to solicit comments and opinions:

- bringing it up at the weekly Ironic meeting
- bringing it up on IRC
- bringing it up on the mailing list (add [Ironic] to the Subject of the email)

If there are enough core folks at the weekly meeting, after discussing an issue, voting could happen and a decision could be made. The problem with IRC or the weekly meeting is that feedback will only come from the people that are actually present.

To inform (and solicit feedback from) more people about an issue, the preferred process is:

1. bring it up on the mailing list
2. after some period of time has elapsed (and depending on the thread activity), someone should propose a solution via gerrit. (E.g. the person that started the thread if no one else steps up.) The

proposal should be made in the git repository that is associated with the issue. (For instance, this decision process was proposed as a documentation patch to the ironic repository.)

3. In the email thread, dont forget to provide a link to the proposed patch!
4. The discussion then moves to the proposed patch. If this is a big decision, we could declare that some percentage of the cores should vote on it before landing it.

(This process was suggested in an email thread about process for making decisions.)

add support for GMRs to new executables and extending the GMR?

For more information, see the oslo.reports documentation page.

Contributor Vision

Background

During the Rocky Project Teams Gathering (Feburary/March 2018), The contributors in the room at that time took a few minutes to write out each contributors vision of where they see ironic in five years time.

After everyone had a chance to spend a few minutes writing, we went around the room and gave every contributor the chance to read their vision and allow other contributors to ask questions to better understand what each individual contributor wrote. While we were doing that, we also took time to capture the common themes.

This entire exercise did result in some laughs and a common set of words, and truly helped to ensure that the entire team proceeded to use the same words to describe various aspects as the sessions progressed during the week. We also agreed that we should write a shared vision, to have something to reference and remind us of where we want to go as a community.

Rocky Vision: For 2022-2023

Common Themes

Below is an entirely unscientific summary of common themes that arose during the discussion among fourteen contributors.

- Contributors picked a time between 2020, and 2023.
- 4 Contributors foresee ironic being the leading Open Source baremetal deployment technology
- 2 Contributors foresee ironic reaching feature parity with Nova.
- 2 Contributors foresee users moving all workloads to the cloud
- 1 Contributor foresees Kubernetes and Container integration being the major focus of Bare Metal as a Service further down the road.
- 2 Contributors foresee greater composible hardware being more common.
- 1 Contributor foresees ironic growing into or supporting CMDBs.
- 2 Contributors foresee that features are more micro-service oriented.

- 2 Contributors foresee that ironic supported all of the possible baremetal management needs
- 1 Contributor foresees standalone use being more common.
- 2 Contributors foresee the ironics developer community growing
- 2 Contributors foresee that auto-discovery will be more common.
- 2 Contributors foresee ironic being used for devices beyond servers, such as lightbulbs, IOT, etc.

Vision Statement

The year is 2022. Were meeting to plan the Z release of Ironic. We stopped to reflect upon the last few years of Ironics growth, how we had come such a long way to become the defacto open source baremetal deployment technology. How we had grown our use cases, and support for consumers such as containers, and users who wished to managed specialized fleets of composed machines.

New contributors and their different use cases have brought us closer to parity with virtual machines. Everyday were gaining word of more operators adopting the ironic communitys CMDB integration to leverage hardware discovery. Weve heard of operators deploying racks upon racks of new hardware by just connecting the power and network cables, and from there the operators have discovered time to write the worlds greatest operator novel with the time saved in commissioning new racks of hardware.

Time has brought us closer and taught us to be more collaborative across the community, and we look forward to our next release together.

Comparison to the 2018 OpenStack Technical Vision

In late-2018, the OpenStack Technical composed a technical vision of what OpenStack clouds should look like. While every component differs, and cloudy interactions change dramatically the closer to physical hardware one gets, there are a few areas where Ironic could use some improvement.

This list is largely for the purposes of help wanted. It is also important to note that Ironic as a project has a vision document for itself.

The Pillars of Cloud - Self Service

- Ironics mechanisms and tooling are low level infrastructure mechanisms and as such there has never been a huge emphasis or need on making Ironic be capable of offering direct multi-tenant interaction. Most users interact with the bare metal managed by Ironic via Nova, which abstracts away many of these issues. Eventually, we should offer direct multi-tenancy which is not oriented towards admin-only.

Design Goals - Built-in Reliability and Durability

- Ironic presently considers in-flight operations as failed upon the restart of a controller that was previously performing a task, because we do not know the current status of the task upon re-start. In some cases, this makes sense, but potentially requires administrative intervention in the worst of cases. In a perfect universe, Ironic conductors would validate their perception, in case tasks actually finished.

Design Goals - Graphical User Interface

- While a graphical interface was developed for Horizon in the form of ironic-ui, currently ironic-ui receives only minimal housekeeping. As Ironic has evolved, ironic-ui is stuck on version *1.34* and knows nothing of our evolution since. Ironic ultimately needs a contributor with sufficient time to pick up `ironic-ui` or to completely replace it as a functional and customizable user interface.

The following pages describe the architecture of the Bare Metal service and may be helpful to anyone working on or with the service, but are written primarily for developers.

System Architecture

High Level description

An Ironic deployment will be composed of the following components:

- An admin-only RESTful API service, by which privileged users, such as cloud operators and other services within the cloud control plane, may interact with the managed bare metal servers.
- A Conductor service, which does the bulk of the work. Functionality is exposed via the API service. The Conductor and API services communicate via RPC.
- A Database and DB API for storing the state of the Conductor and Drivers.
- A Deployment Ramdisk or Deployment Agent, which provide control over the hardware which is not available remotely to the Conductor. A ramdisk should be built which contains one of these agents, eg. with diskimage-builder. This ramdisk can be booted on-demand.

  Note: The agent is never run inside a tenant instance.

Drivers

The internal driver API provides a consistent interface between the Conductor service and the driver implementations. A driver is defined by a *hardware type* deriving from the AbstractHardwareType class, defining supported *hardware interfaces*. See *Enabling drivers and hardware types* for a more detailed explanation. See *Pluggable Drivers* for an explanation on how to write new hardware types and interfaces.

Driver-Specific Periodic Tasks

Drivers may run their own periodic tasks, i.e. actions run repeatedly after a certain amount of time. Such a task is created by using the periodic decorator on an interface method. For example

```
from futurist import periodics

class FakePower(base.PowerInterface):
    @periodics.periodic(spacing=42)
    def task(self, manager, context):
        pass  # do something
```

Here the `spacing` argument is a period in seconds for a given periodic task. For example spacing=5 means every 5 seconds.

Starting with the Yoga cycle, there is also a new decorator *ironic.conductor.periodics.node_periodic()* to create periodic tasks that handle nodes. See *deploy steps documentation* for an example.

Driver-Specific Steps

Drivers may have specific steps that may need to be executed or offered to a user to execute in order to perform specific configuration tasks.

These steps should ideally be located on the management interface to enable consistent user experience of the hardware type. What should be avoided is duplication of existing interfaces such as the deploy interface to enable vendor specific cleaning or deployment steps.

Message Routing

Each Conductor registers itself in the database upon start-up, and periodically updates the timestamp of its record. Contained within this registration is a list of the drivers which this Conductor instance supports. This allows all services to maintain a consistent view of which Conductors and which drivers are available at all times.

Based on their respective driver, all nodes are mapped across the set of available Conductors using a consistent hashing algorithm. Node-specific tasks are dispatched from the API tier to the appropriate conductor using conductor-specific RPC channels. As Conductor instances join or leave the cluster, nodes may be remapped to different Conductors, thus triggering various driver actions such as take-over or clean-up.

Developing New Notifications

Ironic notifications are events intended for consumption by external services. Notifications are sent to these services over a message bus by oslo.messagings Notifier class. For more information about configuring notifications and available notifications, see *Notifications*.

Ironic also has a set of base classes that assist in clearly defining the notification itself, the payload, and the other fields not auto-generated by oslo (level, event_type and publisher_id). Below describes how to use these base classes to add a new notification to ironic.

Adding a new notification to ironic

To add a new notification to ironic, a new versioned notification class should be created by subclassing the NotificationBase class to define the notification itself and the NotificationPayloadBase class to define which fields the new notification will contain inside its payload. You may also define a schema to allow the payload to be automatically populated by the fields of an ironic object. Heres an example:

```
# The ironic object whose fields you want to use in your schema
@base.IronicObjectRegistry.register
class ExampleObject(base.IronicObject):
    # Version 1.0: Initial version
    VERSION = '1.0'
    fields = {
        'id': fields.IntegerField(),
        'uuid': fields.UUIDField(),
        'a_useful_field': fields.StringField(),
        'not_useful_field': fields.StringField()
    }

# A class for your new notification
@base.IronicObjectRegistry.register
class ExampleNotification(notification.NotificationBase):
    # Version 1.0: Initial version
    VERSION = '1.0'
    fields = {
        'payload': fields.ObjectField('ExampleNotifPayload')
    }

# A class for your notification's payload
@base.IronicObjectRegistry.register
class ExampleNotifPayload(notification.NotificationPayloadBase):
    # Schemas are optional. They just allow you to reuse other objects'
    # fields by passing in that object and calling populate_schema with
    # a kwarg set to the other object.
    SCHEMA = {
        'a_useful_field': ('example_obj', 'a_useful_field')
    }

    # Version 1.0: Initial version
    VERSION = '1.0'
```

(continues on next page)

(continued from previous page)

```
    fields = {
        'a_useful_field': fields.StringField(),
        'an_extra_field': fields.StringField(nullable=True)
    }
```

Note that both the payload and notification classes are oslo versioned objects. Modifications to these require a version bump so that consumers of notifications know when the notifications have changed.

SCHEMA defines how to populate the payload fields. Its an optional attribute that subclasses may use to easily populate notifications with data from other objects.

It is a dictionary where every key value pair has the following format:

```
<payload_field_name>: (<data_source_name>,
                       <field_of_the_data_source>)
```

The `<payload_field_name>` is the name where the data will be stored in the payload object; this field has to be defined as a field of the payload. The `<data_source_name>` shall refer to name of the parameter passed as kwarg to the payloads `populate_schema()` call and this object will be used as the source of the data. The `<field_of_the_data_source>` shall be a valid field of the passed argument.

The SCHEMA needs to be applied with the `populate_schema()` call before the notification can be emitted.

The value of the `payload.<payload_field_name>` field will be set by the `<data_source_name>.<field_of_the_data_source>` field. The `<data_source_name>` will not be part of the payload object internal or external representation.

Payload fields that are not set by the SCHEMA can be filled in the same way as in any versioned object.

Then, to create a payload, you would do something like the following. Note that if you choose to define a schema in the SCHEMA class variable, you must populate the schema by calling `populate_schema(example_obj=my_example_obj)` before emitting the notification is allowed:

```
my_example_obj = ExampleObject(id=1,
                               a_useful_field='important',
                               not_useful_field='blah')

# an_extra_field is optional since it's not a part of the SCHEMA and is a
# nullable field in the class fields
my_notify_payload = ExampleNotifyPayload(an_extra_field='hello')
# populate the schema with the ExampleObject fields
my_notify_payload.populate_schema(example_obj=my_example_obj)
```

You then create the notification with the oslo required fields (event_type, publisher_id, and level, all sender fields needed by oslo that are defined in the ironic notification base classes) and emit it:

```
notify = ExampleNotification(
    event_type=notification.EventType(object='example_obj',
        action='do_something', status=fields.NotificationStatus.START),
    publisher=notification.NotificationPublisher(
        service='ironic-conductor',
```

(continues on next page)

(continued from previous page)

```
        host='hostname01'),
    level=fields.NotificationLevel.DEBUG,
    payload=my_notify_payload)
notify.emit(context)
```

When specifying the event_type, `object` will specify the object being acted on, `action` will be a string describing what action is being performed on that object, and `status` will be one of start, end, error, or success. start and end are used to indicate when actions that are not immediate begin and succeed. success is used to indicate when actions that are immediate succeed. error is used to indicate when any type of action fails, regardless of whether its immediate or not. As a result of specifying these parameters, event_type will be formatted as `baremetal.<object>.<action>.<status>` on the message bus.

This example will send the following notification over the message bus:

```
{
    "priority": "debug",
    "payload":{
        "ironic_object.namespace":"ironic",
        "ironic_object.name":"ExampleNotifyPayload",
        "ironic_object.version":"1.0",
        "ironic_object.data":{
             "a_useful_field":"important",
             "an_extra_field":"hello"
        }
    },
    "event_type":"baremetal.example_obj.do_something.start",
    "publisher_id":"ironic-conductor.hostname01"
 }
```

About OSProfiler

OSProfiler is an OpenStack cross-project profiling library. Its API provides different ways to add a new trace point. Trace points contain two messages (start and stop). Messages like below are sent to a collector:

```
{
    "name": <point_name>-(start|stop),
    "base_id": <uuid>,
    "parent_id": <uuid>,
    "trace_id": <uuid>,
    "info": <dict>
}
```

The fields are defined as follows:

`base_id` - <uuid> that is same for all trace points that belong to one trace. This is used to simplify the process of retrieving all trace points (related to one trace) from the collector.

`parent_id` - <uuid> of parent trace point.

`trace_id` - <uuid> of current trace point.

`info` - the dictionary that contains user information passed when calling profiler start() & stop() methods.

The profiler uses ceilometer as a centralized collector. Two other alternatives for ceilometer are pure MongoDB driver and Elasticsearch.

A notifier is setup to send notifications to ceilometer using oslo.messaging and ceilometer API is used to retrieve all messages related to one trace.

OSProfiler has entry point that allows the user to retrieve information about traces and present it in HTML/JSON using CLI.

For more details see OSProfiler Cross-project profiling library.

How to Use OSProfiler with Ironic in Devstack

To use or test OSProfiler in ironic, the user needs to setup Devstack with OSProfiler and ceilometer. In addition to the setup described at *Deploying Ironic with DevStack*, the user needs to do the following:

Add the following to `localrc` to enable OSProfiler and ceilometer:

```
enable_plugin panko https://opendev.org/openstack/panko
enable_plugin ceilometer https://opendev.org/openstack/ceilometer
enable_plugin osprofiler https://opendev.org/openstack/osprofiler

# Enable the following services
CEILOMETER_NOTIFICATION_TOPICS=notifications,profiler
ENABLED_SERVICES+=,ceilometer-acompute,ceilometer-acentral
ENABLED_SERVICES+=,ceilometer-anotification,ceilometer-collector
ENABLED_SERVICES+=,ceilometer-alarm-evaluator,ceilometer-alarm-notifier
ENABLED_SERVICES+=,ceilometer-api
```

Run stack.sh.

Once Devstack environment is setup, edit `ironic.conf` to set the following profiler options and restart ironic services:

```
[profiler]
enabled = True
hmac_keys = SECRET_KEY # default value used across several OpenStack projects
trace_sqlalchemy = True
```

In order to trace ironic using OSProfiler, use openstackclient to run baremetal commands with `--os-profile SECRET_KEY`.

For example, the following will cause a <trace-id> to be printed after node list:

```
$ openstack --os-profile SECRET_KEY baremetal node list
```

Output of the above command will include the following:

```
Trace ID: <trace-id>
Display trace with command:
osprofiler trace show --html <trace-id>
```

The trace results can be seen using this command:

```
$ osprofiler trace show --html <trace-id>
```

The trace results can be saved in a file with `--out file-name` option:

```
$ osprofiler trace show --html <trace-id> --out trace.html
```

The trace results show the time spent in ironic-api, ironic-conductor, and db calls. More detailed db tracing is enabled if `trace_sqlalchemy` is set to true.

References

- OSProfiler Cross-project profiling library
- *Deploying Ironic with DevStack*

Rolling Upgrades

The ironic (ironic-api and ironic-conductor) services support rolling upgrades, starting with a rolling upgrade from the Ocata to the Pike release. This describes the design of rolling upgrades, followed by notes for developing new features or modifying an IronicObject.

Design

Rolling upgrades between releases

Ironic follows the release-cycle-with-intermediary release model. The releases are semantic-versioned, in the form <major>.<minor>.<patch>. We refer to a `named release` of ironic as the release associated with a development cycle like Pike.

In addition, ironic follows the standard deprecation policy, which says that the deprecation period must be at least three months and a cycle boundary. This means that there will never be anything that is both deprecated *and* removed between two named releases.

Rolling upgrades will be supported between:

- named release N to N+1 (starting with N == Ocata)
- any named release to its latest revision, containing backported bug fixes. Because those bug fixes can contain improvements to the upgrade process, the operator should patch the system before upgrading between named releases.
- most recent named release N (and semver releases newer than N) to master. As with the above bullet point, there may be a bug or a feature introduced on a master branch, that we want to remove before publishing a named release. Deprecation policy allows to do this in a 3 month time frame. If the feature was included and removed in intermediate releases, there should be a release note added, with instructions on how to do a rolling upgrade to master from an affected release or release span. This would typically instruct the operator to upgrade to a particular intermediate release, before upgrading to master.

Rolling upgrade process

Ironic supports rolling upgrades as described in the *upgrade guide*.

The upgrade process will cause the ironic services to be running the `FromVer` and `ToVer` releases in this order:

0. Upgrade ironic code and run database schema migrations via the `ironic-dbsync upgrade` command.
1. Upgrade code and restart ironic-conductor services, one at a time.
2. Upgrade code and restart ironic-api services, one at a time.
3. Unpin API, RPC and object versions so that the services can now use the latest versions in `ToVer`. This is done via updating the configuration option described below in *API, RPC and object version pinning* and then restarting the services. ironic-conductor services should be restarted first, followed by the ironic-api services. This is to ensure that when new functionality is exposed on the unpinned API service (via API micro version), it is available on the backend.

step	ironic-api	ironic-conductor
0	all FromVer	all FromVer
1.1	all FromVer	some FromVer, some ToVer-pinned
1.2	all FromVer	all ToVer-pinned
2.1	some FromVer, some ToVer-pinned	all ToVer-pinned
2.2	all ToVer-pinned	all ToVer-pinned
3.1	all ToVer-pinned	some ToVer-pinned, some ToVer
3.2	all ToVer-pinned	all ToVer
3.3	some ToVer-pinned, some ToVer	all ToVer
3.4	all ToVer	all ToVer

Policy for changes to the DB model

The policy for changes to the DB model is as follows:

- Adding new items to the DB model is supported.
- The dropping of columns or tables and corresponding objects fields is subject to ironics deprecation policy. But its alembic script has to wait one more deprecation period, otherwise an `unknown column` exception will be thrown when `FromVer` services access the DB. This is because **`ironic-dbsync upgrade`** upgrades the DB schema but `FromVer` services still contain the dropped field in their SQLAlchemy DB model.
- An `alembic.op.alter_column()` to rename or resize a column is not allowed. Instead, split it into multiple operations, with one operation per release cycle (to maintain compatibility with an old SQLAlchemy model). For example, to rename a column, add the new column in release N, then remove the old column in release N+1.
- Some implementations of SQLs `ALTER TABLE`, such as adding foreign keys in PostgreSQL, may impose table locks and cause downtime. If the change cannot be avoided and the impact is significant (e.g. the table can be frequently accessed and/or store a large dataset), these cases must be mentioned in the release notes.

API, RPC and object version pinning

For the ironic services to be running old and new releases at the same time during a rolling upgrade, the services need to be able to handle different API, RPC and object versions.

This versioning is handled via the configuration option: `[DEFAULT]/pin_release_version`. It is used to pin the API, RPC and IronicObject (e.g., Node, Conductor, Chassis, Port, and Portgroup) versions for all the ironic services.

The default value of empty indicates that ironic-api and ironic-conductor will use the latest versions of API, RPC and IronicObjects. Its possible values are releases, named (e.g. `ocata`) or sem-versioned (e.g. `7.0`).

Internally, in common/release_mappings.py, ironic maintains a mapping that indicates the API, RPC and IronicObject versions associated with each release. This mapping is maintained manually.

During a rolling upgrade, the services using the new release will set the configuration option value to be the name (or version) of the old release. This will indicate to the services running the new release, which API, RPC and object versions that they should be compatible with, in order to communicate with the services using the old release.

Handling API versions

When the (newer) service is pinned, the maximum API version it supports will be the pinned version which the older service supports (as described above at *API, RPC and object version pinning*). The ironic-api service returns HTTP status code 406 for any requests with API versions that are higher than this maximum version.

Handling RPC versions

ConductorAPI.__init__() sets the `version_cap` variable to the desired (latest or pinned) RPC API version and passes it to the `RPCClient` as an initialization parameter. This variable is then used to determine the maximum requested message version that the `RPCClient` can send.

Each RPC call can customize the request according to this `version_cap`. The *Ironic RPC versions* section below has more details about this.

Handling IronicObject versions

Internally, ironic services deal with IronicObjects in their latest versions. Only at these boundaries, when the IronicObject enters or leaves the service, do we deal with object versioning:

- getting objects from the database: convert to latest version
- saving objects to the database: if pinned, save in pinned version; else save in latest version
- serializing objects (to send over RPC): if pinned, send pinned version; else send latest version
- deserializing objects (receiving objects from RPC): convert to latest version

The ironic-api service also has to handle API requests/responses based on whether or how a feature is supported by the API version and object versions. For example, when the ironic-api service is pinned, it

can only allow actions that are available to the objects pinned version, and cannot allow actions that are only available for the latest version of that object.

To support this:

- All the database tables (SQLAlchemy models) of the IronicObjects have a column named `version`. The value is the version of the object that is saved in the database.
- The method `IronicObject.get_target_version()` returns the target version. If pinned, the pinned version is returned. Otherwise, the latest version is returned.
- The method `IronicObject.convert_to_version()` converts the object into the target version. The target version may be a newer or older version than the existing version of the object. The bulk of the work is done in the helper method `IronicObject._convert_to_version()`. Subclasses that have new versions redefine this to perform the actual conversions.

In the following,

- The old release is `FromVer`; it uses version 1.14 of a Node object.
- The new release is `ToVer`. It uses version 1.15 of a Node object this has a deprecated `extra` field and a new `meta` field that replaces `extra`.
- db_obj[meta] and db_obj[extra] are the database representations of those node fields.

Getting objects from the database (API/conductor < DB)

Both ironic-api and ironic-conductor services read values from the database. These values are converted to IronicObjects via the method `IronicObject._from_db_object()`. This method always returns the IronicObject in its latest version, even if it was in an older version in the database. This is done regardless of the service being pinned or not.

Note that if an object is converted to a later version, that IronicObject will retain any changes (in its `_changed_fields` field) resulting from that conversion. This is needed in case the object gets saved later, in the latest version.

For example, if the node in the database is in version 1.14 and has db_obj[extra] set:

- a `FromVer` service will get a Node with node.extra = db_obj[extra] (and no knowledge of node.meta since it doesnt exist)
- a `ToVer` service (pinned or unpinned), will get a Node with:
  - node.meta = db_obj[extra]
  - node.extra = None
  - node._changed_fields = [meta, extra]

Saving objects to the database (API/conductor > DB)

The version used for saving IronicObjects to the database is determined as follows:

- For an unpinned service, the object is saved in its latest version. Since objects are always in their latest version, no conversions are needed.
- For a pinned service, the object is saved in its pinned version. Since objects are always in their latest version, the object needs to be converted to the pinned version before being saved.

The method `IronicObject.do_version_changes_for_db()` handles this logic, returning a dictionary of changed fields and their new values (similar to the existing `oslo.versionedobjects.VersionedObject.obj_get_changes()`). Since we do not keep track internally, of the database version of an object, the objects `version` field will always be part of these changes.

The *Rolling upgrade process* (at step 3.1) ensures that by the time an object can be saved in its latest version, all services are running the newer release (although some may still be pinned) and can handle the latest object versions.

An interesting situation can occur when the services are as described in step 3.1. It is possible for an IronicObject to be saved in a newer version and subsequently get saved in an older version. For example, a `ToVer` unpinned conductor might save a node in version 1.5. A subsequent request may cause a `ToVer` pinned conductor to replace and save the same node in version 1.4!

Sending objects via RPC (API/conductor -> RPC)

When a service makes an RPC request, any IronicObjects that are sent as part of that request are serialized into entities or primitives via `IronicObjectSerializer.serialize_entity()`. The version used for objects being serialized is as follows:

- For an unpinned service, the object is serialized to its latest version. Since objects are always in their latest version, no conversions are needed.
- For a pinned service, the object is serialized to its pinned version. Since objects are always in their latest version, the object is converted to the pinned version before being serialized. The converted object includes changes that resulted from the conversion; this is needed so that the service at the other end of the RPC request has the necessary information if that object will be saved to the database.

Receiving objects via RPC (API/conductor <- RPC)

When a service receives an RPC request, any entities that are part of the request need to be deserialized (via `oslo.versionedobjects.VersionedObjectSerializer.deserialize_entity()`). For entities that represent IronicObjects, we want the deserialization process (via `IronicObjectSerializer._process_object()`) to result in IronicObjects that are in their latest version, regardless of the version they were sent in and regardless of whether the receiving service is pinned or not. Again, any objects that are converted will retain the changes that resulted from the conversion, useful if that object is later saved to the database.

For example, a `FromVer` ironic-api could issue an `update_node()` RPC request with a node in version 1.4, where node.extra was changed (so node._changed_fields = [extra]). This node will be serialized in version 1.4. The receiving `ToVer` pinned ironic-conductor deserializes it and converts it to version 1.5.

The resulting node will have node.meta set (to the changed value from node.extra in v1.4), node.extra = None, and node._changed_fields = [meta, extra].

When developing a new feature or modifying an IronicObject

When adding a new feature or changing an IronicObject, they need to be coded so that things work during a rolling upgrade.

The following describe areas where the code may need to be changed, as well as some points to keep in mind when developing code.

ironic-api

During a rolling upgrade, the new, pinned ironic-api is talking to a new conductor that might also be pinned. There may also be old ironic-api services. So the new, pinned ironic-api service needs to act like it was the older service:

- New features should not be made available, unless they are somehow totally supported in the old and new releases. Pinning the API version is in place to handle this.
  - If, for whatever reason, the API version pinning doesnt prevent a request from being handled that cannot or should not be handled, it should be coded so that the response has HTTP status code 406 (Not Acceptable). This is the same response to requests that have an incorrect (old) version specified.

Ironic RPC versions

When the signature (arguments) of an RPC method is changed or new methods are added, the following needs to be considered:

- The RPC version must be incremented and be the same value for both the client (`ironic/conductor/rpcapi.py`, used by ironic-api) and the server (`ironic/conductor/manager.py`, used by ironic-conductor). It should also be updated in `ironic/common/release_mappings.py`.
- Until there is a major version bump, new arguments of an RPC method can only be added as optional. Existing arguments cannot be removed or changed in incompatible ways with the method in older RPC versions.
- ironic-api (client-side) sets a version cap (by passing the version cap to the constructor of oslo_messaging.RPCClient). This pinning is in place during a rolling upgrade when the `[DEFAULT]/pin_release_version` configuration option is set.
- New RPC methods are not available when the service is pinned to the older release version. In this case, the corresponding REST API function should return a server error or implement alternative behaviours.
- Methods which change arguments should run `client.can_send_version()` to see if the version of the request is compatible with the version cap of the RPC Client. Otherwise the request needs to be created to work with a previous version that is supported.

- ironic-conductor (server-side) should tolerate older versions of requests in order to keep working during the rolling upgrade process. The behaviour of ironic-conductor will depend on the input parameters passed from the client-side.
- Old methods can be removed only after they are no longer used by a previous named release.

Object versions

When subclasses of `ironic.objects.base.IronicObject` are modified, the following needs to be considered:

- Any change of fields or change in signature of remotable methods needs a bump of the object version. The object versions are also maintained in `ironic/common/release_mappings.py`.
- New objects must be added to `ironic/common/release_mappings.py`. Also for the first releases they should be excluded from the version check by adding their class names to the `NEW_MODELS` list in `ironic/cmd/dbsync.py`.
- The arguments of remotable methods (methods which are remoted to the conductor via RPC) can only be added as optional. They cannot be removed or changed in an incompatible way (to the previous release).
- Field types cannot be changed. Instead, create a new field and deprecate the old one.
- There is a unit test that generates the hash of an object using its fields and the signatures of its remotable methods. Objects that have a version bump need to be updated in the expected_object_fingerprints dictionary; otherwise this test will fail. A failed test can also indicate to the developer that their change(s) to an object require a version bump.
- When new version objects communicate with old version objects and when reading or writing to the database, `ironic.objects.base.IronicObject._convert_to_version()` will be called to convert objects to the target version. Objects should implement their own ._convert_to_version() to remove or alter fields which were added or changed after the target version:

```
def _convert_to_version(self, target_version,
                        remove_unavailable_fields=True):
    """Convert to the target version.

    Subclasses should redefine this method, to do the conversion of the
    object to the target version.

    Convert the object to the target version. The target version may be
    the same, older, or newer than the version of the object. This is
    used for DB interactions as well as for serialization/deserialization.

    The remove_unavailable_fields flag is used to distinguish these two
    cases:

    1) For serialization/deserialization, we need to remove the␣
↪unavailable
       fields, because the service receiving the object may not know about
       these fields. remove_unavailable_fields is set to True in this␣
↪case.
```

(continues on next page)

(continued from previous page)

```
    2) For DB interactions, we need to set the unavailable fields to their
       appropriate values so that these fields are saved in the DB. (If
       they are not set, the VersionedObject magic will not know to
       save/update them to the DB.) remove_unavailable_fields is set to
       False in this case.

    :param target_version: the desired version of the object
    :param remove_unavailable_fields: True to remove fields that are
        unavailable in the target version; set this to True when
        (de)serializing. False to set the unavailable fields to␣
↪appropriate
        values; set this to False for DB interactions.
```

This method must handle:

- converting from an older version to a newer version
- converting from a newer version to an older version
- making sure, when converting, that you take into consideration other object fields that may have been affected by a field (value) only available in a newer version. For example, if field new is only available in Node version 1.5 and Node.affected = Node.new+3, when converting to 1.4 (an older version), you may need to change the value of Node.affected too.

Online data migrations

The `ironic-dbsync online_data_migrations` command will perform online data migrations.

Keep in mind the *Policy for changes to the DB model*. Future incompatible changes in SQLAlchemy models, like removing or renaming columns and tables can break rolling upgrades (when ironic services are run with different release versions simultaneously). It is forbidden to remove these database resources when they may still be used by the previous named release.

When creating new Alembic migrations which modify existing models, make sure that any new columns default to NULL. Test the migration out on a non-empty database to make sure that any new constraints dont cause the database to be locked out for normal operations.

You can find an overview on what DDL operations may cause downtime in https://dev.mysql.com/doc/refman/5.7/en/innodb-create-index-overview.html. (You should also check older, widely deployed InnoDB versions for issues.) In the case of PostgreSQL, adding a foreign key may lock a whole table for writes.

Make sure to add a release note if there are any downtime-related concerns.

Backfilling default values, and migrating data between columns or between tables must be implemented inside an online migration script. A script is a database API method (added to `ironic/db/api.py` and `ironic/db/sqlalchemy/api.py`) which takes two arguments:

- context: an admin context
- max_count: this is used to limit the query. It is the maximum number of objects to migrate; >= 0. If zero, all the objects will be migrated.

It returns a two-tuple:

- the total number of objects that need to be migrated, at the start of the method, and
- the number of migrated objects.

In this method, the version column can be used to select and update old objects.

The method name should be added to the list of `ONLINE_MIGRATIONS` in `ironic/cmd/dbsync.py`.

The method should be removed in the next named release after this one.

After online data migrations are completed and the SQLAlchemy models no longer contain old fields, old columns can be removed from the database. This takes at least 3 releases, since we have to wait until the previous named release no longer contains references to the old schema. Before removing any resources from the database by modifying the schema, make sure that your implementation checks that all objects in the affected tables have been migrated. This check can be implemented using the version column.

ironic-dbsync upgrade command

The `ironic-dbsync upgrade` command first checks that the versions of the objects are compatible with the (new) release of ironic, before it will make any DB schema changes. If one or more objects are not compatible, the upgrade will not be performed.

This check is done by comparing the objects `version` field in the database with the expected (or supported) versions of these objects. The supported versions are the versions specified in `ironic.common.release_mappings.RELEASE_MAPPING`. The newly created tables cannot pass this check and thus have to be excluded by adding their object class names (e.g. `Node`) to `ironic.cmd.dbsync.NEW_MODELS`.

Role Based Access Control - Testing

The Role Based Access control testing is a minor departure from the Ironic standard pattern of entirely python based unit testing. In part this was done for purposes of speed and to keep the declaration of the test context.

This also lended itself to be very useful due to the nature of A/B testing which is requried to properly migrate the Ironic project from a project scoped universe where an `admin project` is utilized as the authenticating factor coupled with two custom roles, `baremetal_admin`, and `baremetal_observer`.

As a contributor looking back after getting a over a thousand additional tests in place using this method, it definitely helped the speed at which these were created, and then ported to support additional.

How these tests work

These tests execute API calls through the API layer, using the appropriate verb and header, which settings to prevent the `keystonemiddleware` from intercepting and replacing the headers were passing. Ultimately this is a feature, and it helps quite a bit.

The second aspect of how this works is were mocking the conductor RPC `get_topic_for` and `get_random_topic_for` methods. These calls raise Temporary Unavailable, since trying to execute the entire interaction into the conductor is moderately pointless because all policy enforement is located with-in the API layer.

At the same time wiring everything up to go from API to conductor code and back would have been a heavier lift. As such, the tests largely look for one of the following error codes.

- 200 - Got the item from the API - This is an database driven interaction.
- 201 - Created - This is databaes driven interaction. These are rare.
- 204 - Accepted - This is a database driven interaction. These are rare.
- **403 - Forbidden - This tells us the policy worked as expected where** access was denied.
- **404 - NotFound - This is typically when objects were not found. Before** Ironic becomes scope aware, these are generally only in the drivers API endpoints behavior. In System scope aware Project scoped configuration, i.e. later RBAC tests, this will become the dominant response for project scoped users as responding with a 403 if they could be an owner or lessee would provide insight into the existence of a node.
- **503 - Service Unavailable - In the context of our tests, we expect this** when a request *has* been successfully authenticated and would have been sent along to the conductor.

How to make changes or review these tests?

The tests cycle through the various endpoints, and repeating patterns are clearly visible. Typically this means a given endpoint is cycled through with the same basic test using slightly different parameters such as different authentication parameters. When it comes to system scope aware tests supporting node `owners` and `lessee`, these tests will cycle a little more with slightly different attributes as the operation is not general against a shared common node, but different nodes.

Some tests will test body contents, or attributes. some will validate the number of records returned. This is important later with `owner` and `lessee` having slightly different views of the universe.

Some general rules apply

- Admins can do things, at least as far as their scope or rights apply. Remember: owner and lessee admins are closer to System scoped Admin Members.
- Members can do some things, but not everything
- Readers can always read, but as we get into sensitive data later on such as fields containing infrastucture internal addresses, these values will become hidden and additional tests will examine this.
- Third party, or external/other Admins will find nothing but sadness in empty lists, 403, 404, or even 500 errors.

What is/will be tested?

The idea is to in essence test as much as possible, however as these tests Role Based Access Control related capabilities will come in a series of phases, styles vary a little.

The first phase is `"legacy"`. In essence these are partially programatically generated and then human reviewed and values populated with expected values.

The second phase is remarkably similar to `legacy`. It is the safety net where we execute the `legacy` tests with the updated `oslo.policy` configuration to help enforce scopes. These tests will intentionally begin to fail in phase three.

The third phase is the implementation of System scope awareness for the API. In this process, as various portions of the API are made system scope aware. The `legacy` tests are marked as `deprecated` which

signals to the second phase test sequences that they are **expected** to fail. New `system scoped` tests are also implemented which are matched up by name to the `legacy` tests. The major difference being some header values, and a user with a `member` role in the `system` scope now has some rights.

The forth phase, is implementaiton of `owner` and `lessee` aware project scoping. The testing approach is similar, however it is much more of a shotgun approach. We test what we know should work, and what know should not work, but we do not have redundant testing for each role as `admin` users are also `members`, and since the policy rules are designed around thresholds of access, it just made no sense to run the same test for admin and members, where member was the threshold. These thresholds will vary with the proposed default policy. The forth scope also tests a third party external admin as a negative test to ensure that we are also denying access to resources appropriately.

These pages contain information for PTLs, cross-project liaisons, and core reviewers.

Releasing Ironic Projects

Since the responsibility for releases will move between people, we document that process here.

A full list of projects that ironic manages is available in the governance site.

Who is responsible for releases?

The current PTL is ultimately responsible for making sure code gets released. They may choose to delegate this responsibility to a liaison, which is documented in the cross-project liaison wiki.

Anyone may submit a release request per the process below, but the PTL or liaison must +1 the request for it to be processed.

Release process

Releases are managed by the OpenStack release team. The release process is documented in the Project Team Guide.

What do we have to release?

The ironic project has a number of deliverables under its governance. The ultimate source of truth for this is projects.yaml in the governance repository. These deliverables have varying release models, and these are defined in the deliverables YAML files in the releases repository.

In general, ironic deliverables follow the cycle-with-intermediary release model.

Non-client libraries

The following deliverables are non-client libraries:

- ironic-lib
- metalsmith
- sushy

Client libraries

The following deliverables are client libraries:

- python-ironicclient
- python-ironic-inspector-client

Normal release

The following deliverables are Neutron plugins:

- networking-baremetal
- networking-generic-switch

The following deliverables are Horizon plugins:

- ironic-ui

The following deliverables are Tempest plugins:

- ironic-tempest-plugin

The following deliverables are tools:

- ironic-python-agent-builder

The following deliverables are services, or treated as such:

- bifrost
- ironic
- ironic-inspector
- ironic-prometheus-exporter
- ironic-python-agent

Manual release

The ironic-staging-drivers follows a different procedure, see Releasing ironic-staging-drivers.

Independent

The following deliverables are released independently:

- sushy-tools
- tenks
- virtualbmc

Not released

The following deliverables do not need to be released:

- ironic-inspector-specs
- ironic-specs

Bugfix branches

The following projects have `bugfix/X.Y` branches in addition to standard openstack `stable/NAME` branches:

- bifrost
- ironic
- ironic-inspector
- ironic-python-agent

They are also released on a regular cadence as opposed to on-demand, namely three times a release cycle (roughly a release every 2 months). One of the releases corresponds to the coordinated OpenStack released and receives a `stable/NAME` branch. The other two happen during the cycle and receive a `bugfix/X.Y` branch, where `X.Y` consists of the major and the minor component of the version (e.g. `bugfix/8.1` for 8.1.0).

To leave some version space for releases from these branches, releases of these projects from the master branch always increase either the major or the minor version.

Currently releases from bugfix branches cannot be automated and must be done by the release team manually.

Things to do before releasing

- Review the unreleased release notes, if the project uses them. Make sure they follow our *standards*, are coherent, and have proper grammar. Combine release notes if necessary (for example, a release note for a feature and another release note to add to that feature may be combined).
- For ironic releases only, not ironic-inspector releases: if any new API microversions have been added since the last release, update the REST API version history (`doc/source/contributor/webapi-version-history.rst`) to indicate that they were part of the new release.
- To support rolling upgrades, add this new release version (and release name if it is a named release) into `ironic/common/release_mappings.py`:
  - in `RELEASE_MAPPING` make a copy of the `master` entry, and rename the first `master` entry to the new semver release version.
  - If this is a named release, add a `RELEASE_MAPPING` entry for the named release. Its value should be the same as that of the latest semver one (that you just added above).

    It is important to do this before a stable/<release> branch is made (or if the grenade switch is made to use the latest release from stable as the old release). Otherwise, once it is made, CI (the grenade job that tests new-release -> master) will fail.
- Check for any open patches that are close to be merged or release critical.

  This usually includes important bug fixes and/or features that wed like to release, including the related documentation.

How to propose a release

The steps that lead to a release proposal are mainly manual, while proposing the release itself is almost a 100% automated process, accomplished by following the next steps:

- Clone the openstack/releases repository. This is where deliverables are tracked and all the automation resides.
  - Under the `deliverables` directory you can see yaml files for each deliverable (i.e. subproject) grouped by release cycles.
  - The `_independent` directory contains yaml files for deliverables that are not bound to (official) cycles (e.g. ironic-python-agent-builder).
- To check the changes were about to release we can use the tox environment `list-unreleased-changes`, with this syntax:

```
tox -e venv -- list-unreleased-changes <series> <deliverable>
```

  The `series` argument is a release series (i.e. master or train, not stable/ussuri or stable/train).

  For example, assuming were in the main directory of the releases repository, to check the changes in the ussuri series for ironic-python-agent type:

```
tox -e venv -- list-unreleased-changes ussuri openstack/ironic-python-
↪agent
```

- To update the deliverable file for the new release, we use a scripted process in the form of a tox environment called `new-release`.

To get familiar with it and see all the options, type:

```
tox -e venv -- new-release -h
```

Now, based on the list of changes we found in the precedent step, and the release notes, we need to decide on whether the next version will be major, minor (feature) or patch (bugfix).

Note that in this case `series` is a code name (train, ussuri), not a branch. That is also valid for the current development branch (master) that takes the code name of the future stable release, for example if the future stable release code name is wallaby, we need to use wallaby as `series`.

The `--stable-branch argument` is used only for branching in the end of a cycle, independent projects are not branched this way though.

The `--intermediate-branch` option is used to create an intermediate bugfix branch following the new release model for ironic projects.

To propose the release, use the script to update the deliverable file, then commit the change, and propose it for review.

For example, to propose a minor release for ironic in the master branch (current development branch), considering that the code name of the future stable release is wallaby, use:

```
tox -e venv -- new-release -v wallaby ironic feature
```

Remember to use a meaningful topic, usually using the name of the deliverable, the new version and the branch, if applicable.

A good commit message title should also include the same, for example Release ironic 1.2.3 for ussuri

- As an optional step, we can use `tox -e list-changes` to double-check the changes before submitting them for review.

  Also `tox -e validate` (it might take a while to run based on the number of changes) does some some sanity-checks, but since everything is scripted, there shouldnt be any issue.

  All the scripts are designed and maintained by the release team; in case of questions or doubts or if any errors should arise, you can reach to them in the IRC channel `#openstack-release`; all release liaisons should be present there.

- After the change is up for review, the PTL or a release liaison will have to approve it before it can get approved by the release team. Then, it will be processed automatically by zuul.

Things to do after releasing

When a release is done that results in a stable branch

When a release is done that results in a stable branch for the project, several changes need to be made.

The release automation will push a number of changes that need to be approved. This includes:

- In the new stable branch:
  - a change to point `.gitreview` at the branch
  - a change to update the upper constraints file used by `tox`

- In the master branch:
  - updating the release notes RST to include the new branch.

    The generated RST does not include the version range in the title, so we typically submit a follow-up patch to do that. An example of this patch is here.
  - update the *templates* in *.zuul.yaml* or *zuul.d/project.yaml*.

    The update is necessary to use the job for the next release *openstack-python3-<next_release>-jobs*. An example of this patch is here.

We need to submit patches for changes in the stable branch to:

- update the ironic devstack plugin to point at the branched tarball for IPA. An example of this patch is here.
- set appropriate defaults for `TEMPEST_BAREMETAL_MIN_MICROVERSION` and `TEMPEST_BAREMETAL_MAX_MICROVERSION` in `devstack/lib/ironic` to make sure that unsupported API tempest tests are skipped on stable branches. E.g. patch 495319.

We need to submit patches for changes on master to:

- to support rolling upgrades, since the release was a named release, we need to make these changes. Note that we need to wait until *after* the switch in grenade is made to test the latest release (N) with master (e.g. for stable/queens). Doing these changes sooner after the ironic release and before the switch when grenade is testing the prior release (N-1) with master, will cause the tests to fail. (You may want to ask/remind infra/qa team, as to when they will do this switch.)
  - In `ironic/common/release_mappings.py`, delete any entries from `RELEASE_MAPPING` associated with the oldest named release. Since we support upgrades between adjacent named releases, the master branch will only support upgrades from the most recent named release to master.
  - remove any DB migration scripts from `ironic.cmd.dbsync.ONLINE_MIGRATIONS` and remove the corresponding code from ironic. (These migration scripts are used to migrate from an old release to this latest release; they shouldnt be needed after that.)

When a release is done that results in a bugfix branch

In this case the release management only creates a change to point `.gitreview` at the branch, `tox.ini` is not modified.

After the release:

- update the Tempest microversions as explained above.
- the CI needs additional configuration, so that Zuul knows which branch to take jobs definitions from. See the following examples:
  - ironic 18.1
  - ironic-inspector 10.7
  - ironic-python-agent 8.1

Ironic Tempest plugin

As **ironic-tempest-plugin** is branchless, we need to submit a patch adding stable jobs to its master branch. Example for Queens.

Bifrost

Bifrost needs to be updated to install dependencies using the stable branch. Example for Victoria. The upper constraints file referenced in `scripts/install-deps.sh` needs to be updated to the new release.

For all releases

For all releases, whether or not it results in a stable branch:

- update the specs repo to mark any specs completed in the release as implemented.
- remove any -2s on patches that were blocked until after the release.

Ironic Governance Structure

The ironic project manages a number of repositories that contribute to our mission. The full list of repositories that ironic manages is available in the governance site.

What belongs in ironic governance?

For a repository to be part of the Ironic project:

- It must comply with the TCs rules for a new project.
- It must not be intended for use with only a single vendors hardware. A library that implements a standard to manage hardware from multiple vendors (such as IPMI or redfish) is okay.
- It must align with Ironics mission statement.

Lack of contributor diversity is a chicken-egg problem, and as such a repository where only a single company is contributing is okay, with the hope that other companies will contribute after joining the ironic project.

Repositories that are no longer maintained should be pruned from governance regularly.

Proposing a new project to ironic governance

Bring the proposal to the ironic weekly meeting to discuss with the team.

Ironics State Machine

The content has been migrated, please see *Bare Metal State Machine*.

8.1.2 Writing Drivers

Ironics community includes many hardware vendors who contribute drivers that enable more advanced functionality when Ironic is used in conjunction with that hardware. To do this, the Ironic developer community is committed to standardizing on a Python Driver API that meets the common needs of all hardware vendors, and evolving this API without breaking backwards compatibility. However, it is sometimes necessary for driver authors to implement functionality - and expose it through the REST API - that can not be done through any existing API.

To facilitate that, we also provide the means for API calls to be passed through ironic and directly to the driver. Some guidelines on how to implement this are provided below. Driver authors are strongly encouraged to talk with the developer community about any implementation using this functionality.

Pluggable Drivers

Ironic supports a pluggable driver model. This allows contributors to easily add new drivers, and operators to use third-party drivers or write their own. A driver is built at runtime from a *hardware type* and *hardware interfaces*. See *Enabling drivers and hardware types* for a detailed explanation of these concepts.

Hardware types and interfaces are loaded by the `ironic-conductor` service during initialization from the setuptools entrypoints `ironic.hardware.types` and `ironic.hardware.interfaces.<INTERFACE>` where `<INTERFACE>` is an interface type (for example, `deploy`). Only hardware types listed in the configuration option `enabled_hardware_types` and interfaces listed in configuration options `enabled_<INTERFACE>_interfaces` are loaded. A complete list of hardware types available on the system may be found by enumerating this entrypoint by running the following python script:

```
#!/usr/bin/env python

import pkg_resources as pkg
print [p.name for p in pkg.iter_entry_points("ironic.hardware.types") if not␣
↪p.name.startswith("fake")]
```

A list of drivers enabled in a running Ironic service may be found by issuing the following command against that API end point:

```
baremetal driver list
```

Writing a hardware type

A hardware type is a Python class, inheriting *ironic.drivers.hardware_type.AbstractHardwareType* and listed in the setuptools entry point `ironic.hardware.types`. Most of the real world hardware types inherit *ironic.drivers.generic.GenericHardware* instead. This helper class provides useful implementations for interfaces that are usually the same for all hardware types, such as `deploy`.

The minimum required interfaces are:

- *boot* that specifies how to boot ramdisks and instances on the hardware. A generic `pxe` implementation is provided by the `GenericHardware` base class.
- *deploy* that orchestrates the deployment. A few common implementations are provided by the `GenericHardware` base class.

  As of the Rocky release, a deploy interface should decorate its deploy method to indicate that it is a deploy step. Conventionally, the deploy method uses a priority of 100.

  ```
  @ironic.drivers.base.deploy_step(priority=100)
  def deploy(self, task):
  ```

  Note: Most of the hardware types should not override this interface.

- *power* implements power actions for the hardware. These common implementations may be used, if supported by the hardware:
  - *ironic.drivers.modules.ipmitool.IPMIPower*
  - *ironic.drivers.modules.redfish.power.RedfishPower*

  Otherwise, you need to write your own implementation by subclassing *ironic.drivers.base.PowerInterface* and providing missing methods.

  Note: Power actions in Ironic are blocking - methods of a power interface should not return until the power action is finished or errors out.

- *management* implements additional out-of-band management actions, such as setting a boot device. A few common implementations exist and may be used, if supported by the hardware:
  - *ironic.drivers.modules.ipmitool.IPMIManagement*
  - *ironic.drivers.modules.redfish.management.RedfishManagement*

  Some hardware types, such as `snmp` do not support out-of-band management. They use the fake implementation in *ironic.drivers.modules.fake.FakeManagement* instead.

  Otherwise, you need to write your own implementation by subclassing *ironic.drivers.base.ManagementInterface* and providing missing methods.

Combine the interfaces in a hardware type by populating the lists of supported interfaces. These lists are prioritized, with the most preferred implementation first. For example:

```
class MyHardware(generic.GenericHardware):

    @property
    def supported_management_interfaces(self):
        """List of supported management interfaces."""
        return [MyManagement, ipmitool.IPMIManagement]

    @property
    def supported_power_interfaces(self):
        """List of supported power interfaces."""
        return [MyPower, ipmitool.IPMIPower]
```

Note: In this example, all interfaces, except for `management` and `power` are taken from the `GenericHardware` base class.

Finally, give the new hardware type and new interfaces human-friendly names and create entry points for them in the `setup.cfg` file:

```
ironic.hardware.types =
    my-hardware = ironic.drivers.my_hardware:MyHardware
ironic.hardware.interfaces.power =
    my-power = ironic.drivers.modules.my_hardware:MyPower
ironic.hardware.interfaces.management =
    my-management = ironic.drivers.modules.my_hardware:MyManagement
```

Deploy and clean steps

Significant parts of the bare metal functionality is implemented via *deploy steps* or *clean steps*. See *Developing deploy and clean steps* for information on how to write them.

Supported Drivers

For a list of supported drivers (those that are continuously tested on every upstream commit) please consult the *drivers page*.

Vendor Methods

This document is a quick tutorial on writing vendor specific methods to a driver.

The first thing to note is that the Ironic API supports two vendor endpoints: A driver vendor passthru and a node vendor passthru.

- The `VendorInterface` allows hardware types to expose a custom top-level functionality which is not specific to a Node. For example, lets say the driver *ipmi* exposed a method called *authentication_types* that would return what are the authentication types supported. It could be accessed via the Ironic API like:

```
GET http://<address>:<port>/v1/drivers/ipmi/vendor_passthru/
↪authentication_types
```

> **Warning:** The Bare Metal API currently only allows to use driver passthru for the default `vendor` interface implementation for a given hardware type. This limitation will be lifted in the future.

- The node vendor passthru allows drivers to expose custom functionality on per-node basis. For example the same driver *ipmi* exposing a method called *send_raw* that would send raw bytes to the BMC, the method also receives a parameter called *raw_bytes* which the value would be the bytes to be sent. It could be accessed via the Ironic API like:

```
POST {'raw_bytes': '0x01 0x02'} http://<address>:<port>/v1/nodes/<node␣
↪UUID>/vendor_passthru/send_raw
```

Writing Vendor Methods

Writing a custom vendor method in Ironic should be simple. The first thing to do is write a class inheriting from the VendorInterface class:

```
class ExampleVendor(VendorInterface)

    def get_properties(self):
        return {}

    def validate(self, task, **kwargs):
        pass
```

The *get_properties* is a method that all driver interfaces have, it should return a dictionary of <property>:<description> telling in the description whether that property is required or optional so the node can be manageable by that driver. For example, a required property for a *ipmi* driver would be *ipmi_address* which is the IP address or hostname of the node. We are returning an empty dictionary in our example to make it simpler.

The *validate* method is responsible for validating the parameters passed to the vendor methods. Ironic will not introspect into what is passed to the drivers, its up to the developers writing the vendor method to validate that data.

Lets extend the *ExampleVendor* class to support two methods, the *authentication_types* which will be exposed on the driver vendor passthru endpoint; And the *send_raw* method that will be exposed on the node vendor passthru endpoint:

```
class ExampleVendor(VendorInterface)

    def get_properties(self):
        return {}

    def validate(self, task, method, **kwargs):
        if method == 'send_raw':
```

(continues on next page)

(continued from previous page)

```
            if 'raw_bytes' not in kwargs:
                raise MissingParameterValue()

    @base.driver_passthru(['GET'], async_call=False)
    def authentication_types(self, context, **kwargs):
        return {"types": ["NONE", "MD5", "MD2"]}

    @base.passthru(['POST'])
    def send_raw(self, task, **kwargs):
        raw_bytes = kwargs.get('raw_bytes')
        ...
```

Thats it!

Writing a node or driver vendor passthru method is pretty much the same, the only difference is how you decorate the methods and the first parameter of the method (ignoring self). A method decorated with the *@passthru* decorator should expect a Task object as first parameter and a method decorated with the *@driver_passthru* decorator should expect a Context object as first parameter.

Both decorators accept these parameters:

- http_methods: A list of what the HTTP methods supported by that vendor function. To know what HTTP method that function was invoked with, a *http_method* parameter will be present in the *kwargs*. Supported HTTP methods are *POST*, *PUT*, *GET* and *PATCH*.
- method: By default the method name is the name of the python function, if you want to use a different name this parameter is where this name can be set. For example:

```
@passthru(['PUT'], method="alternative_name")
def name(self, task, **kwargs):
    ...
```

- description: A string containing a nice description about what that method is supposed to do. Defaults to (empty string).
- async_call: A boolean value to determine whether this method should run asynchronously or synchronously. Defaults to True (Asynchronously).

Note: This parameter was previously called async.

The node vendor passthru decorator (*@passthru*) also accepts the following parameter:

- require_exclusive_lock: A boolean value determining whether this method should require an exclusive lock on a node between validate() and the beginning of method execution. For synchronous methods, the lock on the node would also be kept for the duration of method execution. Defaults to True.

Warning: Please avoid having a synchronous method for slow/long-running operations **or** if the method does talk to a BMC; BMCs are flaky and very easy to break.

Warning: Each asynchronous request consumes a worker thread in the `ironic-conductor` process. This can lead to starvation of the thread pool, resulting in a denial of service.

Give the new vendor interface implementation a human-friendly name and create an entry point for it in the `setup.cfg`:

```
ironic.hardware.interfaces.vendor =
    example = ironic.drivers.modules.example:ExampleVendor
```

Finally, add it to the list of supported vendor interfaces for relevant hardware types, for example:

```
class ExampleHardware(generic.GenericHardware):
    ...

    @property
    def supported_vendor_interfaces(self):
        return [example.ExampleVendor]
```

Backwards Compatibility

There is no requirement that changes to a vendor method be backwards compatible. However, for your users sakes, we highly recommend that you do so.

If you are changing the exceptions being raised, you might want to ensure that the same HTTP code is being returned to the user.

For non-backwards compatibility, please make sure you add a release note that indicates this.

Developing BIOS Interface

To support a driver specific BIOS interface it is necessary to create a class inheriting from the `BIOSInterface` class:

```
from ironic.drivers import base

class ExampleBIOS(base.BIOSInterface):

    def get_properties(self):
        return {}

    def validate(self, task):
        pass
```

See *Pluggable Drivers* for a detailed explanation of hardware type and interface.

The `get_properties` and `validate` are methods that all driver interfaces have. The hardware interface that supports BIOS settings should also implement the following three methods:

- Implement a method named `cache_bios_settings`. This method stores BIOS settings to the `bios_settings` table during cleaning operations and updates the `bios_settings` table when `apply_configuration` or `factory_reset` are successfully called.

```
from ironic.drivers import base

driver_client = importutils.try_import('driver.client')

class ExampleBIOS(base.BIOSInterface):
    def __init__(self):
        if driver_client is None:
            raise exception.DriverLoadError(
                driver=self.__class__.__name__,
                reason=_("Unable to import driver library"))

    def cache_bios_settings(self, task):
        node_id = task.node.id
        node_info = driver_common.parse_driver_info(task.node)
        settings = driver_client.get_bios_settings(node_info)
        create_list, update_list, delete_list, nochange_list = (
            objects.BIOSSettingList.sync_node_setting(settings))

        if len(create_list) > 0:
            objects.BIOSSettingList.create(
                task.context, node_id, create_list)
        if len(update_list) > 0:
            objects.BIOSSettingList.save(
                task.context, node_id, update_list)
        if len(delete_list) > 0:
            delete_names = []
            for setting in delete_list:
                delete_names.append(setting.name)
            objects.BIOSSettingList.delete(
                task.context, node_id, delete_names)
```

Note: `driver.client` is vendor specific library to control and manage the bare metal hardware, for example: python-dracclient, sushy.

- Implement a method named `factory_reset`. This method needs to use the `clean_step` decorator. It resets BIOS settings to factory default on the given node. It calls `cache_bios_settings` automatically to update existing `bios_settings` table once successfully executed.

```
class ExampleBIOS(base.BIOSInterface):

    @base.clean_step(priority=0)
    def factory_reset(self, task):
        node_info = driver_common.parse_driver_info(task.node)
        driver_client.reset_bios_settings(node_info)
```

- Implement a method named `apply_configuration`. This method needs to use the clean_step decorator. It takes the given BIOS settings and applies them on the node. It also calls `cache_bios_settings` automatically to update existing `bios_settings` table after successfully applying given settings on the node.

```
class ExampleBIOS(base.BIOSInterface):

    @base.clean_step(priority=0, argsinfo={
        'settings': {
            'description': (
                'A list of BIOS settings to be applied'
            ),
            'required': True
        }
    })
    def apply_configuration(self, task, settings):
        node_info = driver_common.parse_driver_info(task.node)
        driver_client.apply_bios_settings(node_info, settings)
```

The `settings` parameter is a list of BIOS settings to be configured. for example:

```
[
  {
    "setting name":
      {
        "name": "String",
        "value": "String"
      }
  },
  {
    "setting name":
      {
        "name": "String",
        "value": "String"
      }
  },
  ...
]
```

Third Party Continuous Integration

Note: This document is a work-in-progress. Unfilled sections will be worked in follow-up patchsets. This version is to get a basic outline and index done so that we can then build on it. (krtaylor)

This document provides tips and guidelines for third-party driver developers setting up their continuous integration test systems.

CI Architecture Overview

Requirements Cookbook

Sizing

Infrastructure

This section describes what changes youll need to make to a your CI system to add an ironic job.

jenkins changes

nodepool changes

neutron changes

pre-test hook

cleanup hook

Ironic

Hardware Pool Management

Problem

If you are using actual hardware as target machines for your CI testing then the problem of two jobs trying to use the name target arises. If you have one target machine and a maximum number of one jobs running on your ironic pipeline at a time, then you wont run into this problem. However, one target may not handle the load of ironics daily patch submissions.

Solutions

Zuul v3

Molten Iron

molteniron is a tool that allows you to reserve hardware from a pool at the last minute to use in your job. Once finished testing, you can unreserve the hardware making it available for the next test job.

Tips and Tricks

Optimize Run Time

Image Server

Other References

Developing deploy and clean steps

Deploy steps basics

To support customized deployment step, implement a new method in an interface class and use the decorator `deploy_step` defined in `ironic/drivers/base.py`. For example, we will implement a `do_nothing` deploy step in the `AgentDeploy` class.

```
from ironic.drivers.modules import agent

class AgentDeploy(agent.AgentDeploy):

    @base.deploy_step(priority=200, argsinfo={
        'test_arg': {
            'description': (
                "This is a test argument."
            ),
            'required': True
        }
    })
    def do_nothing(self, task, **kwargs):
        return None
```

If you want to completely replace the deployment procedure, but still have the agent up and running, inherit `CustomAgentDeploy`:

```
from ironic.drivers.modules import agent

class AgentDeploy(agent.CustomAgentDeploy):

    def validate(self, task):
        super().validate(task)
        # ... custom validation

    @base.deploy_step(priority=80)
    def my_write_image(self, task, **kwargs):
        pass  # ... custom image writing

    @base.deploy_step(priority=70)
    def my_configure_bootloader(self, task, **kwargs):
        pass  # ... custom bootloader configuration
```

After deployment of the baremetal node, check the updated deploy steps:

```
baremetal node show $node_ident -f json -c driver_internal_info
```

The above command outputs the `driver_internal_info` as following:

```
{
  "driver_internal_info": {
    ...
    "deploy_steps": [
      {
        "priority": 200,
        "interface": "deploy",
        "step": "do_nothing",
        "argsinfo":
          {
            "test_arg":
              {
                "required": True,
                "description": "This is a test argument."
              }
          }
      },
      {
        "priority": 100,
        "interface": "deploy",
        "step": "deploy",
        "argsinfo": null
      }
    ],
    "deploy_step_index": 1
  }
}
```

In-band deploy steps (deploy steps that are run inside the ramdisk) have to be implemented in a custom IPA hardware manager. All in-band deploy steps must have priorities between 41 and 99, see *Agent steps* for details.

Clean steps basics

Clean steps are written similarly to deploy steps, but are executed during *cleaning*. Steps with priority > 0 are executed during automated cleaning, all steps can be executed explicitly during manual cleaning. Unlike deploy steps, clean steps are commonly found in these interfaces:

bios Steps that apply BIOS settings, see *Implementing BIOS settings*.

deploy Steps that undo the effect of deployment (e.g. erase disks).

management Additional steps that use the nodes BMC, such as out-of-band firmware update or BMC reset.

raid Steps that build or tear down RAID, see *Implementing RAID*.

Note: When designing a new step for your driver, try to make it consistent with existing steps on other drivers.

Just as deploy steps, in-band clean steps have to be implemented in a custom IPA hardware manager.

Asynchronous steps

If the step returns `None`, ironic assumes its execution is finished and proceeds to the next step. Many steps are executed asynchronously; in this case you need to inform ironic that the step is not finished. There are several possibilities:

Combined in-band and out-of-band step

If your step starts as out-of-band and then proceeds as in-band (i.e. inside the agent), you only need to return `CLEANWAIT/DEPLOYWAIT` from the step.

```
from ironic.drivers import base
from ironic.drivers.modules import agent
from ironic.drivers.modules import agent_base
from ironic.drivers.modules import agent_client
from ironic.drivers.modules import deploy_utils

class MyDeploy(agent.CustomAgentDeploy):
    ...

    @base.deploy_step(priority=80)
    def my_deploy(self, task):
        ...
        return deploy_utils.get_async_step_return_state(task.node)

    # Usually you can use a more high-level pattern:

    @base.deploy_step(priority=60)
    def my_deploy2(self, task):
        new_step = {'interface': 'deploy',
                    'step': 'my_deploy2',
                    'args': {...}}
        client = agent_client.get_client(task)
        return agent_base.execute_step(task, new_step, 'deploy',
                                       client=client)
```

Warning: This approach only works for steps implemented on a `deploy` interface that inherits agent deploy.

Execution on reboot

Some steps are executed out-of-band, but require a reboot to complete. Use the following pattern:

```
from ironic.drivers import base
from ironic.drivers.modules import deploy_utils

class MyManagement(base.ManagementInterface):
    ...

    @base.clean_step(priority=0)
    def my_action(self, task):
        ...

        # Tell ironic that...
        deploy_utils.set_async_step_flags(
            node,
            # ... we're waiting for IPA to come back after reboot
            reboot=True,
            # ... the current step is done
            skip_current_step=True)

        return deploy_utils.reboot_to_finish_step(task)
```

Polling for completion

Finally, you may want to poll the BMC until the operation is complete. Often enough, this also involves a reboot. In this case you can use the *ironic.conductor.periodics.node_periodic()* decorator to create a periodic task that operates on relevant nodes:

```
from ironic.common import states
from ironic.common import utils
from ironic.conductor import periodics
from ironic.drivers import base
from ironic.drivers.modules import deploy_utils

_STATUS_CHECK_INTERVAL = ...  # better use a configuration option

class MyManagement(base.ManagementInterface):
    ...

    @base.clean_step(priority=0)
    def my_action(self, task):
        ...

        reboot_required = ...  # your step may or may not need rebooting

        # Make this node as running my_action. Often enough you will store
        # some useful data rather than a boolean flag.
```

(continues on next page)

(continued from previous page)

```
        utils.set_node_nested_field(task.node, 'driver_internal_info',
                                    'in_my_action', True)

        # Tell ironic that...
        deploy_utils.set_async_step_flags(
            node,
            # ... we're waiting for IPA to come back after reboot
            reboot=reboot_required,
            # ... the current step shouldn't be entered again
            skip_current_step=True,
            # ... we'll be polling until the step is done
            polling=True)

        if reboot_required:
            return deploy_utils.reboot_to_finish_step(task)

    @periodics.node_periodic(
        purpose='checking my action status',
        spacing=_STATUS_CHECK_INTERVAL,
        filters={
            # Skip nodes that already have a lock
            'reserved': False,
            # Only consider nodes that are waiting for cleaning or failed
            # on timeout.
            'provision_state_in': [states.CLEANWAIT, states.CLEANFAIL],
        },
        # Load driver_internal_info from the database on listing
        predicate_extra_fields=['driver_internal_info'],
        # Only consider nodes with in_my_action
        predicate=lambda n: n.driver_internal_info.get('in_my_action'),
    )
    def check_my_action(self, task, manager, context):
        if not needs_actions():  # insert your checks here
            return

        task.upgrade_lock()

        ...  # do any required updates

        # Drop the flag so that this node is no longer considered
        utils.pop_node_nested_field(task.node, 'driver_internal_info',
                                    'in_my_action')
```

Note that creating a `task` involves an additional database query, so you want to avoid creating them for too many nodes in your periodic tasks. Instead:

- Try to use precise `filters` to filter out nodes on the database level. Using `reserved` and `provision_state`/`provision_state_in` are recommended in most cases. See *ironic.db.api.Connection.get_nodeinfo_list()* for a list of possible filters.
- Use `predicate` to filter on complex fields such as `driver_internal_info`. Predicates are

checked before tasks are created.

Implementing RAID

RAID is implemented via deploy and clean steps in the `raid` interfaces. By convention they have the following signatures:

```
from ironic.drivers import base

class MyRAID(base.RAIDInterface):

    @base.clean_step(priority=0, abortable=False, argsinfo={
        'create_root_volume': {
            'description': (
                'This specifies whether to create the root volume. '
                'Defaults to `True`.'
            ),
            'required': False
        },
        'create_nonroot_volumes': {
            'description': (
                'This specifies whether to create the non-root volumes. '
                'Defaults to `True`.'
            ),
            'required': False
        },
        'delete_existing': {
            'description': (
                'Setting this to `True` indicates to delete existing RAID '
                'configuration prior to creating the new configuration. '
                'Default value is `False`.'
            ),
            'required': False,
        }
    })
    def create_configuration(self, task, create_root_volume=True,
                             create_nonroot_volumes=True,
                             delete_existing=False):
        pass

    @base.clean_step(priority=0)
    @base.deploy_step(priority=0)
    def delete_configuration(self, task):
        pass

    @base.deploy_step(priority=0,
                      argsinfo=base.RAID_APPLY_CONFIGURATION_ARGSINFO)
    def apply_configuration(self, task, raid_config,
                            create_root_volume=True,
                            create_nonroot_volumes=False,
```

(continues on next page)

(continued from previous page)

```
                                delete_existing=False):
        pass
```

Notes:

- `create_configuration` only works as a clean step, during deployment `apply_configuration` is used instead.
- `apply_configuration` accepts the target RAID configuration explicitly, while `create_configuration` uses the nodes `target_raid_config` field.
- Priorities default to 0 since RAID should not be built by default.

Implementing BIOS settings

BIOS is implemented via deploy and clean steps in the `raid` interfaces. By convention they have the following signatures:

```
from ironic.drivers import base

_APPLY_CONFIGURATION_ARGSINFO = {
    'settings': {
        'description': (
            'A list of BIOS settings to be applied'
        ),
        'required': True
    }
}

class MyBIOS(base.BIOSInterface):

    @base.clean_step(priority=0)
    @base.deploy_step(priority=0)
    @base.cache_bios_settings
    def factory_reset(self, task):
        pass

    @base.clean_step(priority=0, argsinfo=_APPLY_CONFIGURATION_ARGSINFO)
    @base.deploy_step(priority=0, argsinfo=_APPLY_CONFIGURATION_ARGSINFO)
    @base.cache_bios_settings
    def apply_configuration(self, task, settings):
        pass
```

Notes:

- Both `factory_reset` and `apply_configuration` can be used as deploy and clean steps.
- The `cache_bios_settings` decorator is used to ensure that the settings cached in the ironic database is updated.
- Priorities default to 0 since BIOS settings should not be modified by default.

8.1.3 Testing Network Integration

In order to test the integration between the Bare Metal and Networking services, support has been added to devstack to mimic an external physical switch. Here we include a recommended configuration for devstack to bring up this environment.

Ironic multitenant networking and DevStack

This guide will walk you through using OpenStack Ironic/Neutron with the ML2 `networking-generic-switch` plugin.

Using VMs as baremetal servers

This scenario shows how to setup Devstack to use Ironic/Neutron integration with VMs as baremetal servers and ML2 `networking-generic-switch` that interacts with OVS.

DevStack Configuration

The following is `local.conf` that will setup Devstack with 3 VMs that are registered in ironic. `networking-generic-switch` driver will be installed and configured in Neutron.

```
[[local|localrc]]

# Configure ironic from ironic devstack plugin.
enable_plugin ironic https://opendev.org/openstack/ironic

# Install networking-generic-switch Neutron ML2 driver that interacts with OVS
enable_plugin networking-generic-switch https://opendev.org/openstack/
↪networking-generic-switch

# Add link local info when registering Ironic node
IRONIC_USE_LINK_LOCAL=True

IRONIC_ENABLED_NETWORK_INTERFACES=flat,neutron
IRONIC_NETWORK_INTERFACE=neutron

#Networking configuration
OVS_PHYSICAL_BRIDGE=brbm
PHYSICAL_NETWORK=mynetwork
IRONIC_PROVISION_NETWORK_NAME=ironic-provision
IRONIC_PROVISION_SUBNET_PREFIX=10.0.5.0/24
IRONIC_PROVISION_SUBNET_GATEWAY=10.0.5.1

Q_PLUGIN=ml2
ENABLE_TENANT_VLANS=True
Q_ML2_TENANT_NETWORK_TYPE=vlan
TENANT_VLAN_RANGE=100:150
```

(continues on next page)

(continued from previous page)

```
# Credentials
ADMIN_PASSWORD=password
RABBIT_PASSWORD=password
DATABASE_PASSWORD=password
SERVICE_PASSWORD=password
SERVICE_TOKEN=password
SWIFT_HASH=password
SWIFT_TEMPURL_KEY=password

# Enable Ironic API and Ironic Conductor
enable_service ironic
enable_service ir-api
enable_service ir-cond

# Disable nova novnc service, ironic does not support it anyway.
disable_service n-novnc

# Enable Swift for the direct deploy interface.
enable_service s-proxy
enable_service s-object
enable_service s-container
enable_service s-account

# Disable Horizon
disable_service horizon

# Disable Cinder
disable_service cinder c-sch c-api c-vol

# Disable Tempest
disable_service tempest

# Set glance's default limit to be baremetal image friendly
GLANCE_LIMIT_IMAGE_SIZE_TOTAL=5000

# Swift temp URL's are required for the direct deploy interface.
SWIFT_ENABLE_TEMPURLS=True

# Create 3 virtual machines to pose as Ironic's baremetal nodes.
IRONIC_VM_COUNT=3
IRONIC_BAREMETAL_BASIC_OPS=True

# Enable additional hardware types, if needed.
#IRONIC_ENABLED_HARDWARE_TYPES=ipmi,fake-hardware
# Don't forget that many hardware types require enabling of additional
# interfaces, most often power and management:
#IRONIC_ENABLED_MANAGEMENT_INTERFACES=ipmitool,fake
#IRONIC_ENABLED_POWER_INTERFACES=ipmitool,fake
#IRONIC_DEFAULT_DEPLOY_INTERFACE=direct
```

(continues on next page)

(continued from previous page)

```
# Change this to alter the default driver for nodes created by devstack.
# This driver should be in the enabled list above.
IRONIC_DEPLOY_DRIVER=ipmi

# The parameters below represent the minimum possible values to create
# functional nodes.
IRONIC_VM_SPECS_RAM=1024
IRONIC_VM_SPECS_DISK=10

# Size of the ephemeral partition in GB. Use 0 for no ephemeral partition.
IRONIC_VM_EPHEMERAL_DISK=0

# To build your own IPA ramdisk from source, set this to True
IRONIC_BUILD_DEPLOY_RAMDISK=False

VIRT_DRIVER=ironic

# By default, DevStack creates a 10.0.0.0/24 network for instances.
# If this overlaps with the hosts network, you may adjust with the
# following.
NETWORK_GATEWAY=10.1.0.1
FIXED_RANGE=10.1.0.0/24
FIXED_NETWORK_SIZE=256

# Log all output to files
LOGFILE=$HOME/devstack.log
LOGDIR=$HOME/logs
IRONIC_VM_LOG_DIR=$HOME/ironic-bm-logs
```

8.1.4 Testing Boot-from-Volume

Starting with the Pike release, it is also possible to use DevStack for testing booting from Cinder volumes with VMs.

Ironic Boot-from-Volume with DevStack

This guide shows how to setup DevStack for enabling boot-from-volume feature, which has been supported from the Pike release.

This scenario shows how to setup DevStack to enable nodes to boot from volumes managed by cinder with VMs as baremetal servers.

DevStack Configuration

The following is `local.conf` that will setup DevStack with 3 VMs that are registered in ironic. A volume connector with IQN is created for each node. These connectors can be used to connect volumes created by cinder. The detailed description for DevStack is at *Deploying Ironic with DevStack*.

```
[[local|localrc]]

enable_plugin ironic https://opendev.org/openstack/ironic

IRONIC_STORAGE_INTERFACE=cinder

# Credentials
ADMIN_PASSWORD=password
DATABASE_PASSWORD=password
RABBIT_PASSWORD=password
SERVICE_PASSWORD=password
SERVICE_TOKEN=password
SWIFT_HASH=password
SWIFT_TEMPURL_KEY=password

# Set glance's default limit to be baremetal image friendly
GLANCE_LIMIT_IMAGE_SIZE_TOTAL=5000

# Enable Neutron which is required by Ironic and disable nova-network.
disable_service n-net
disable_service n-novnc
enable_service q-svc
enable_service q-agt
enable_service q-dhcp
enable_service q-l3
enable_service q-meta
enable_service neutron

# Enable Swift for the direct deploy interface.
enable_service s-proxy
enable_service s-object
enable_service s-container
enable_service s-account

# Disable Horizon
disable_service horizon
```

(continues on next page)

(continued from previous page)

```
# Disable Heat
disable_service heat h-api h-api-cfn h-api-cw h-eng

# Swift temp URL's are required for the direct deploy interface.
SWIFT_ENABLE_TEMPURLS=True

# Create 3 virtual machines to pose as Ironic's baremetal nodes.
IRONIC_VM_COUNT=3
IRONIC_BAREMETAL_BASIC_OPS=True
DEFAULT_INSTANCE_TYPE=baremetal

# Enable additional hardware types, if needed.
#IRONIC_ENABLED_HARDWARE_TYPES=ipmi,fake-hardware
# Don't forget that many hardware types require enabling of additional
# interfaces, most often power and management:
#IRONIC_ENABLED_MANAGEMENT_INTERFACES=ipmitool,fake
#IRONIC_ENABLED_POWER_INTERFACES=ipmitool,fake
#IRONIC_DEFAULT_DEPLOY_INTERFACE=direct

# Change this to alter the default driver for nodes created by devstack.
# This driver should be in the enabled list above.
IRONIC_DEPLOY_DRIVER=ipmi

# The parameters below represent the minimum possible values to create
# functional nodes.
IRONIC_VM_SPECS_RAM=1280
IRONIC_VM_SPECS_DISK=10

# Size of the ephemeral partition in GB. Use 0 for no ephemeral partition.
IRONIC_VM_EPHEMERAL_DISK=0

# To build your own IPA ramdisk from source, set this to True
IRONIC_BUILD_DEPLOY_RAMDISK=False

VIRT_DRIVER=ironic

# By default, DevStack creates a 10.0.0.0/24 network for instances.
# If this overlaps with the hosts network, you may adjust with the
# following.
NETWORK_GATEWAY=10.1.0.1
FIXED_RANGE=10.1.0.0/24
FIXED_NETWORK_SIZE=256

# Log all output to files
LOGFILE=$HOME/devstack.log
LOGDIR=$HOME/logs
IRONIC_VM_LOG_DIR=$HOME/ironic-bm-logs
```

After the environment is built, you can create a volume with cinder and request an instance with the

volume to nova:

```
. ~/devstack/openrc

# query the image id of the default cirros image
image=$(openstack image show $DEFAULT_IMAGE_NAME -f value -c id)

# create keypair
ssh-keygen
openstack keypair create --public-key ~/.ssh/id_rsa.pub default

# create volume
volume=$(openstack volume create --image $image --size 1 my-volume -f value -
↪c id)

# spawn instance
openstack server create --flavor baremetal --volume $volume --key-name␣
↪default testing
```

You can also run an integration test that an instance is booted from a remote volume with tempest in the environment:

```
cd /opt/stack/tempest
tox -e all-plugin -- ironic_tempest_plugin.tests.scenario.test_baremetal_boot_
↪from_volume
```

Please note that the storage interface will only indicate errors based upon the state of the node and the configuration present. As such a node does not exclusively have to boot via a remote volume, and as such *validate* actions upon nodes may be slightly misleading. If an appropriate *volume target* is defined, no error should be returned for the boot interface.

8.1.5 Full Ironic Server Python API Reference

ironic

ironic package

Subpackages

ironic.api package

Subpackages

ironic.api.controllers package

Subpackages

ironic.api.controllers.v1 package

Submodules

ironic.api.controllers.v1.allocation module

class ironic.api.controllers.v1.allocation.**AllocationsController**(**args*, ***kwargs*)

Bases: pecan.rest.RestController

REST controller for allocations.

delete(*allocation_ident*)

Delete an allocation.

Parameters
• **allocation_ident** The UUID or name of the allocation.

get_all(*node=None*, *resource_class=None*, *state=None*, *marker=None*, *limit=None*, *sort_key='id'*, *sort_dir='asc'*, *fields=None*, *owner=None*)

Retrieve a list of allocations.

Parameters

- **node** Filter the list of allocations by the node UUID or name.
- **resource_class** The requested resource class for the allocation. Can only be missing when backfilling an allocation (will be set to the nodes "resource_class" in such case).
- **state** Filter the list of allocations by the allocation state, one of "active", "allocating" or "error".
- **marker** The ID of the last-seen item. Use the "limit" parameter to make an initial limited request and use the ID of the last-seen item from the response as the "marker" parameter value in a subsequent limited request.
- **limit** Requests a page size of items. Returns a number of items up to a limit value. Use the "limit" parameter to make an initial limited request and use the ID of the last-seen item from the response as the "marker" parameter value in a subsequent limited request. This value cannot be larger than the "max_limit" option in the "[api]" section of the configuration. If it is higher than "max_limit", only "max-limit" resources will be returned.
- **sort_key** Sorts the response by this attribute value. Default is "id". You can specify multiple pairs of sort key and sort direction query parameters. If you omit the sort direction in a pair, the API uses the natural sorting direction of the server attribute that is provided as the "sort_key".
- **sort_dir** Sorts the response by the requested sort direction. A valid value is "asc" (ascending) or "desc" (descending). Default is "asc". You can specify multiple pairs of sort key and sort direction query parameters. If you omit the sort direction in a pair, the API uses the natural sorting direction of the server attribute that is provided as the "sort_key".
- **fields** One or more fields to be returned in the response.
  For example, the following request returns only the "uuid" and "name" fields for each node:
  ::
  GET /v1/nodes?fields=uuid,name
- **owner** Filter the list of returned allocations, and only return those with the specified owner.

get_one(*allocation_ident*, *fields=None*)

Retrieve information about the given allocation.

Parameters

- **allocation_ident** The UUID or name of the allocation.
- **fields** One or more fields to be returned in the response.
  For example, the following request returns only the "uuid" and "name" fields for each node:
  ::
  GET /v1/nodes?fields=uuid,name

invalid_sort_key_list = ['extra', 'candidate_nodes', 'traits']

patch(*allocation_ident*, *patch*)
Update an existing allocation.

Parameters

- **allocation_ident** The UUID or name of the allocation.
- **patch** A JSON patch document to apply to the allocation.

post(*allocation*)
Create a new allocation.

Parameters

- **allocation** The unique name of the Allocation.

class ironic.api.controllers.v1.allocation.**NodeAllocationController**(**args*, ***kwargs*)

Bases: pecan.rest.RestController

REST controller for allocations.

delete()
Delete an allocation.

get_all(*fields=None*)
Get all allocations.

Parameters

- **fields** One or more fields to be returned in the response.
  For example, the following request returns only the "uuid" and "name" fields for each node:
  ::
  GET /v1/nodes?fields=uuid,name

invalid_sort_key_list = ['extra', 'candidate_nodes', 'traits']

ironic.api.controllers.v1.allocation.**allocation_sanitize**(*allocation*, *fields*)

ironic.api.controllers.v1.allocation.**convert_with_links**(*rpc_allocation*, *fields=None*, *sanitize=True*)

ironic.api.controllers.v1.allocation.**hide_fields_in_newer_versions**(*allocation*)

ironic.api.controllers.v1.allocation.**list_convert_with_links**(*rpc_allocations*, *limit*, *url*, *fields=None*, ***kwargs*)

ironic.api.controllers.v1.bios module

class ironic.api.controllers.v1.bios.**NodeBiosController**(**args*, ***kwargs*)

Bases: pecan.rest.RestController

REST controller for bios.

get_all(*detail=None*, *fields=None*)

List node bios settings.

get_one(*setting_name*)

Retrieve information about the given bios setting.

Parameters **setting_name** Logical name of the setting to retrieve.

ironic.api.controllers.v1.bios.**collection_from_list**(*node_ident*, *bios_settings*, *detail=None*, *fields=None*)

ironic.api.controllers.v1.bios.**convert_with_links**(*rpc_bios*, *node_uuid*, *detail=None*, *fields=None*)

Build a dict containing a bios setting value.

ironic.api.controllers.v1.chassis module

class ironic.api.controllers.v1.chassis.**ChassisController**(**args*, ***kwargs*)

Bases: pecan.rest.RestController

REST controller for Chassis.

delete(*chassis_uuid*)

Delete a chassis.

Parameters **chassis_uuid** UUID of a chassis.

detail(*marker=None*, *limit=None*, *sort_key='id'*, *sort_dir='asc'*)

Retrieve a list of chassis with detail.

Parameters

- **marker** pagination marker for large data sets.
- **limit** maximum number of resources to return in a single result. This value cannot be larger than the value of max_limit in the [api] section of the ironic configuration, or only max_limit resources will be returned.

- **sort_key** column to sort results by. Default: id.
- **sort_dir** direction to sort. asc or desc. Default: asc.

get_all(*marker=None*, *limit=None*, *sort_key='id'*, *sort_dir='asc'*, *fields=None*, *detail=None*)
Retrieve a list of chassis.

Parameters

- **marker** pagination marker for large data sets.
- **limit** maximum number of resources to return in a single result. This value cannot be larger than the value of max_limit in the [api] section of the ironic configuration, or only max_limit resources will be returned.
- **sort_key** column to sort results by. Default: id.
- **sort_dir** direction to sort. asc or desc. Default: asc.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.

get_one(*chassis_uuid*, *fields=None*)
Retrieve information about the given chassis.

Parameters

- **chassis_uuid** UUID of a chassis.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.

invalid_sort_key_list = ['extra']

nodes = <ironic.api.controllers.v1.node.NodesController object>
Expose nodes as a sub-element of chassis

patch(*chassis_uuid*, *patch*)
Update an existing chassis.

Parameters

- **chassis_uuid** UUID of a chassis.
- **patch** a json PATCH document to apply to this chassis.

post(*chassis*)
Create a new chassis.

Parameters **chassis** a chassis within the request body.

ironic.api.controllers.v1.chassis.**convert_with_links**(*rpc_chassis*, *fields=None*, *sanitize=True*)

ironic.api.controllers.v1.chassis.**list_convert_with_links**(*rpc_chassis_list*, *limit*, *url*, *fields=None*, ***kwargs*)

ironic.api.controllers.v1.collection module

ironic.api.controllers.v1.collection.**get_next**(*collection*, *limit*, *url*, *key_field='uuid'*, ***kwargs*)

Return a link to the next subset of the collection.

ironic.api.controllers.v1.collection.**has_next**(*collection*, *limit*)

Return whether collection has more items.

ironic.api.controllers.v1.collection.**list_convert_with_links**(*items*, *item_name*, *limit*, *url*, *fields=None*, *sanitize_func=None*, *key_field='uuid'*, *sanitizer_args=None*, ***kwargs*)

Build a collection dict including the next link for paging support.

Parameters

- **items** List of unsanitized items to include in the collection
- **item_name** Name of dict key for items value
- **limit** Paging limit
- **url** Base URL for building next link
- **fields** Optional fields to use for sanitize function
- **sanitize_func** Optional sanitize function run on each item, item changes will be done in-place
- **key_field** Key name for building next URL
- **sanitizer_args** Dictionary with additional arguments to be passed to the sanitizer.
- **kwargs** other arguments passed to `get_next`

Returns A dict containing `item_name` and `next` values

ironic.api.controllers.v1.conductor module

class ironic.api.controllers.v1.conductor.**ConductorsController**(**args*, ***kwargs*)

Bases: `pecan.rest.RestController`

REST controller for conductors.

get_all(*marker=None*, *limit=None*, *sort_key='id'*, *sort_dir='asc'*, *fields=None*, *detail=None*)

Retrieve a list of conductors.

Parameters

- **marker** pagination marker for large data sets.
- **limit** maximum number of resources to return in a single result. This value cannot be larger than the value of max_limit in the [api] section of the ironic configuration, or only max_limit resources will be returned.
- **sort_key** column to sort results by. Default: id.

- **sort_dir** direction to sort. asc or desc. Default: asc.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.
- **detail** Optional, boolean to indicate whether retrieve a list of conductors with detail.

get_one(*hostname*, *fields=None*)
Retrieve information about the given conductor.

Parameters

- **hostname** hostname of a conductor.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.

invalid_sort_key_list = ['alive', 'drivers']

ironic.api.controllers.v1.conductor.**convert_with_links**(*rpc_conductor*, *fields=None*, *sanitize=True*)

ironic.api.controllers.v1.conductor.**list_convert_with_links**(*rpc_conductors*, *limit*, *url=None*, *fields=None*, ***kwargs*)

ironic.api.controllers.v1.deploy_template module

class ironic.api.controllers.v1.deploy_template.**DeployTemplatesController**(**args*, ***kwargs*)

Bases: pecan.rest.RestController

REST controller for deploy templates.

delete(*template_ident*)
Delete a deploy template.

Parameters **template_ident** UUID or logical name of a deploy template.

get_all(*marker=None*, *limit=None*, *sort_key='id'*, *sort_dir='asc'*, *fields=None*, *detail=None*)
Retrieve a list of deploy templates.

Parameters

- **marker** pagination marker for large data sets.
- **limit** maximum number of resources to return in a single result. This value cannot be larger than the value of max_limit in the [api] section of the ironic configuration, or only max_limit resources will be returned.
- **sort_key** column to sort results by. Default: id.
- **sort_dir** direction to sort. asc or desc. Default: asc.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.
- **detail** Optional, boolean to indicate whether retrieve a list of deploy templates with detail.

get_one(*template_ident*, *fields=None*)
Retrieve information about the given deploy template.

Parameters

- **template_ident** UUID or logical name of a deploy template.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.

invalid_sort_key_list = ['extra', 'steps']

patch(*template_ident*, *patch=None*)
Update an existing deploy template.

Parameters

- **template_ident** UUID or logical name of a deploy template.
- **patch** a json PATCH document to apply to this deploy template.

post(*template*)
Create a new deploy template.

Parameters **template** a deploy template within the request body.

ironic.api.controllers.v1.deploy_template.**convert_steps**(*rpc_steps*)

ironic.api.controllers.v1.deploy_template.**convert_with_links**(*rpc_template*, *fields=None*, *sanitize=True*)
Add links to the deploy template.

ironic.api.controllers.v1.deploy_template.**duplicate_steps**(*name*, *value*)
Argument validator to check template for duplicate steps

ironic.api.controllers.v1.deploy_template.**list_convert_with_links**(*rpc_templates*, *limit*, *fields=None*, ***kwargs*)

ironic.api.controllers.v1.deploy_template.**step_sanitize**(*step*)

ironic.api.controllers.v1.deploy_template.**template_sanitize**(*template*, *fields*)
Removes sensitive and unrequested data.

Will only keep the fields specified in the `fields` parameter.

Parameters **fields** (*list of str*) list of fields to preserve, or `None` to preserve them all

ironic.api.controllers.v1.driver module

class ironic.api.controllers.v1.driver.**DriverPassthruController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

REST controller for driver passthru.

This controller allow vendors to expose cross-node functionality in the Ironic API. Ironic will merely relay the message from here to the specified driver, no introspection will be made in the message body.

methods(*driver_name*)
Retrieve information about vendor methods of the given driver.

Parameters **driver_name** name of the driver.

Returns dictionary with <vendor method name>:<method metadata> entries.

Raises *DriverNotFound* if the driver name is invalid or the driver cannot be loaded.

class ironic.api.controllers.v1.driver.**DriverRaidController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

logical_disk_properties(*driver_name*)
Returns the logical disk properties for the driver.

Parameters
• **driver_name** Name of the driver.

Returns
Success: A dictionary containing the properties that can be mentioned **Failure:** • **UnsupportedDriverExtension (HTTP 400 Bad Request)** Driver %(driver)s does not support %(extension)s (disabled or not implemented). If the driver doesnt support RAID configuration. • **NotAcceptable (HTTP 406 Not Acceptable)** Request not acceptable. If requested version of the API is less than 1.12. • **DriverNotFound (HTTP 404 Not Found)** Could not find the following driver(s) or hardware type(s): %(driver_name)s. If driver is not loaded on any of the conductors.

class ironic.api.controllers.v1.driver.**DriversController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

REST controller for Drivers.

get_all(*type=None*, *detail=None*, *fields=None*)
Retrieve a list of drivers.

get_one(*driver_name*, *fields=None*)
Retrieve a single driver.

properties(*driver_name*)
Retrieve property information of the given driver.

Parameters **driver_name** name of the driver.

Returns dictionary with <property name>:<property description> entries.

Raises *DriverNotFound* **(HTTP 404)** if the driver name is invalid or the driver cannot be loaded.

raid = <ironic.api.controllers.v1.driver.DriverRaidController object>
Expose RAID as a sub-element of drivers

vendor_passthru = <ironic.api.controllers.v1.driver.DriverPassthruController object>

ironic.api.controllers.v1.driver.**convert_with_links**(*name*, *hosts*, *detail=False*, *interface_info=None*, *fields=None*, *sanitize=True*)
Convert driver/hardware type info to a dict.

Parameters

- **name** name of a hardware type.
- **hosts** list of conductor hostnames driver is active on.
- **detail** boolean, whether to include detailed info, such as the type field and default/enabled interfaces fields.
- **interface_info** optional list of dicts of hardware interface info.
- **fields** list of fields to preserve, or `None` to preserve default
- **sanitize** boolean, sanitize driver

Returns dict representing the driver object.

ironic.api.controllers.v1.driver.**driver_sanitize**(*driver*, *fields=None*)

ironic.api.controllers.v1.driver.**hide_fields_in_newer_versions**(*driver*)
This method hides fields that were added in newer API versions.

Certain fields were introduced at certain API versions. These fields are only made available when the requests API version matches or exceeds the versions when these fields were introduced.

ironic.api.controllers.v1.driver.**list_convert_with_links**(*hardware_types*, *detail=False*, *fields=None*)
Convert drivers and hardware types to an API-serializable object.

Parameters

- **hardware_types** dict mapping hardware type names to conductor hostnames.
- **detail** boolean, whether to include detailed info, such as the type field and default/enabled interfaces fields.
- **fields** list of fields to preserve, or `None` to preserve default

Returns an API-serializable driver collection object.

ironic.api.controllers.v1.event module

class ironic.api.controllers.v1.event.**EventsController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

REST controller for Events.

post(*evts*)

ironic.api.controllers.v1.event.**events_valid**(*name*, *value*)
Validator for events

ironic.api.controllers.v1.node module

class ironic.api.controllers.v1.node.**BootDeviceController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

get(*node_ident*)
Get the current boot device for a node.

Parameters **node_ident** the UUID or logical name of a node.

Returns

a json object containing:

boot_device the boot device, one of *ironic.common.boot_devices* or None if it is unknown.

persistent Whether the boot device will persist to all future boots or not, None if it is unknown.

put(*node_ident*, *boot_device*, *persistent=False*)
Set the boot device for a node.

Set the boot device to use on next reboot of the node.

Parameters

- **node_ident** the UUID or logical name of a node.
- **boot_device** the boot device, one of *ironic.common.boot_devices*.
- **persistent** Boolean value. True if the boot device will persist to all future boots, False if not. Default: False.

supported(*node_ident*)
Get a list of the supported boot devices.

Parameters **node_ident** the UUID or logical name of a node.

Returns A json object with the list of supported boot devices.

class ironic.api.controllers.v1.node.**IndicatorAtComponent**(***kwargs*)
Bases: object

class ironic.api.controllers.v1.node.**IndicatorController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

get_all(*node_ident*, ***kwargs*)
Get node hardware components and their indicators.

Parameters **node_ident** the UUID or logical name of a node.

Returns A json object of hardware components (*ironic.common.components*) as keys with indicator IDs (from *get_supported_indicators*) as values.

get_one(*node_ident*, *indicator*)
Get node hardware component indicator and its state.

Parameters

- **node_ident** the UUID or logical name of a node.
- **indicator** Indicator ID (as reported by *get_supported_indicators*).

Returns a dict with the state key and one of mod:*ironic.common.indicator_states* as a value.

put(*node_ident*, *indicator*, *state*)
Set node hardware component indicator to the desired state.

Parameters

- **node_ident** the UUID or logical name of a node.
- **indicator** Indicator ID (as reported by *get_supported_indicators*).
- **state** Indicator state, one of mod:*ironic.common.indicator_states*.

class ironic.api.controllers.v1.node.**InjectNmiController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

put(*node_ident*)
Inject NMI for a node.

Inject NMI (Non Maskable Interrupt) for a node immediately.

Parameters **node_ident** the UUID or logical name of a node.

Raises NotFound if requested version of the API doesnt support inject nmi.

Raises HTTPForbidden if the policy is not authorized.

Raises NodeNotFound if the node is not found.

Raises NodeLocked if the node is locked by another conductor.

Raises UnsupportedDriverExtension if the nodes driver doesnt support management or management.inject_nmi.

Raises InvalidParameterValue when the wrong driver info is specified or an invalid boot device is specified.

Raises MissingParameterValue if missing supplied info.

class ironic.api.controllers.v1.node.**NodeConsoleController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

get(*node_ident*)
Get connection information about the console.

Parameters **node_ident** UUID or logical name of a node.

put(*node_ident*, *enabled*)
Start and stop the node console.

Parameters

- **node_ident** UUID or logical name of a node.
- **enabled** Boolean value; whether to enable or disable the console.

class ironic.api.controllers.v1.node.**NodeHistoryController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

detail_fields = ['uuid', 'created_at', 'severity', 'event_type', 'event', 'conductor', 'user']

get_all(*detail=False*, *marker=None*, *limit=None*)
List node history.

get_one(*event*)
Get a node history entry

standard_fields = ['uuid', 'created_at', 'severity', 'event']

class ironic.api.controllers.v1.node.**NodeMaintenanceController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

delete(*node_ident*)
Remove the node from maintenance mode.

Parameters **node_ident** the UUID or logical name of a node.

put(*node_ident*, *reason=None*)
Put the node in maintenance mode.

Parameters

- **node_ident** the UUID or logical_name of a node.
- **reason** Optional, the reason why its in maintenance.

class ironic.api.controllers.v1.node.**NodeManagementController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

boot_device = <ironic.api.controllers.v1.node.BootDeviceController object>
Expose boot_device as a sub-element of management

indicators = <ironic.api.controllers.v1.node.IndicatorController object>
Expose indicators as a sub-element of management

inject_nmi = <ironic.api.controllers.v1.node.InjectNmiController object>
Expose inject_nmi as a sub-element of management

class ironic.api.controllers.v1.node.**NodeStatesController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

boot_mode(*node_ident*, *target*)
Asynchronous set the boot mode of the node.

Parameters

- **node_ident** the UUID or logical name of a node.
- **target** The desired boot_mode for the node (uefi/bios).

Raises InvalidParameterValue (HTTP 400) if the requested target state is not valid.

Raises NotFound (HTTP 404) if requested version of the API is less than 1.76.

Raises Conflict (HTTP 409) if a node is in adopting state or another transient state.

console = <ironic.api.controllers.v1.node.NodeConsoleController object>
Expose console as a sub-element of states

get(*node_ident*)
List the states of the node.

Parameters **node_ident** the UUID or logical_name of a node.

power(*node_ident*, *target*, *timeout=None*)
Set the power state of the node.

Parameters

- **node_ident** the UUID or logical name of a node.
- **target** The desired power state of the node.
- **timeout** timeout (in seconds) positive integer (> 0) for any power state. None indicates to use default timeout.

Raises ClientSideError (HTTP 409) if a power operation is already in progress.

Raises InvalidStateRequested (HTTP 400) if the requested target state is not valid or if the node is in CLEANING state.

Raises NotAcceptable (HTTP 406) for soft reboot, soft power off or timeout parameter, if requested version of the API is less than 1.27.

Raises Invalid (HTTP 400) if timeout value is less than 1.

provision(*node_ident*, *target*, *configdrive=None*, *clean_steps=None*, *deploy_steps=None*, *rescue_password=None*, *disable_ramdisk=None*)
Asynchronous trigger the provisioning of the node.

This will set the target provision state of the node, and a background task will begin which actually applies the state change. This call will return a 202 (Accepted) indicating the request was accepted and is in progress; the client should continue to GET the status of this node to observe the status of the requested action.

Parameters

- **node_ident** UUID or logical name of a node.
- **target** The desired provision state of the node or verb.
- **configdrive** Optional. A gzipped and base64 encoded configdrive or a dict to build a configdrive from. Only valid when setting provision state to active or rebuild.
- **clean_steps** An ordered list of cleaning steps that will be performed on the node. A cleaning step is a dictionary with required keys interface and step, and optional key args. If specified, the value for args is a keyword variable argument dictionary that is passed to the cleaning step method.:

```
{ 'interface': <driver_interface>,
  'step': <name_of_clean_step>,
  'args': {<arg1>: <value1>, ..., <argn>: <valuen>} }
```

For example (this isnt a real example, this cleaning step doesnt exist):

```
{ 'interface': 'deploy',
  'step': 'upgrade_firmware',
  'args': {'force': True} }
```

This is required (and only valid) when target is clean.

- **deploy_steps** A list of deploy steps that will be performed on the node. A deploy step is a dictionary with required keys interface, step, priority and args. If specified, the value for args is a keyword variable argument dictionary that is passed to the deploy step method.:

```
{ 'interface': <driver_interface>,
  'step': <name_of_deploy_step>,
  'args': {<arg1>: <value1>, ..., <argn>: <valuen>}
  'priority': <integer>}
```

For example (this isnt a real example, this deploy step doesnt exist):

```
{ 'interface': 'deploy',
  'step': 'upgrade_firmware',
  'args': {'force': True},
  'priority': 90 }
```

This is used only when target is active or rebuild and is optional.

- **rescue_password** A string representing the password to be set inside the rescue environment. This is required (and only valid), when target is rescue.
- **disable_ramdisk** Whether to skip booting ramdisk for cleaning.

Raises NodeLocked (HTTP 409) if the node is currently locked.

Raises ClientSideError (HTTP 409) if the node is already being provisioned.

Raises InvalidParameterValue (HTTP 400), if validation of clean_steps, deploy_steps or power driver interface fails.

Raises InvalidStateRequested (HTTP 400) if the requested transition is not possible from the current state.

Raises NodeInMaintenance (HTTP 400), if operation cannot be performed because the node is in maintenance mode.

Raises NoFreeConductorWorker (HTTP 503) if no workers are available.

Raises NotAcceptable (HTTP 406) if the API version specified does not allow the requested state transition or parameters.

raid(*node_ident*, *target_raid_config*)

Set the target raid config of the node.

Parameters

- **node_ident** the UUID or logical name of a node.
- **target_raid_config** Desired target RAID configuration of the node. It may be an empty dictionary as well.

Raises UnsupportedDriverExtension, if the nodes driver doesnt support RAID configuration.

Raises InvalidParameterValue, if validation of target raid config fails.

Raises NotAcceptable, if requested version of the API is less than 1.12.

secure_boot(*node_ident*, *target*)
Asynchronous set the secure_boot state of the node.

Parameters

- **node_ident** the UUID or logical name of a node.
- **target** Should secure_boot be enabled on node (True/False).

Raises InvalidParameterValue (HTTP 400) if the requested target state is not valid.

Raises NotFound (HTTP 404) if requested version of the API is less than 1.76.

Raises Conflict (HTTP 409) if a node is in adopting state.

class ironic.api.controllers.v1.node.**NodeTraitsController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

delete(*trait=None*)
Remove one or all traits from a node.

Parameters **trait** String value; trait to remove from a node, or None. If None, all traits are removed.

get_all()
List node traits.

put(*trait=None*, *body=None*)
Add a trait to a node.

Parameters

- **trait** String value; trait to add to a node, or None. Mutually exclusive with traits. If not None, adds this trait to the node.
- **traits** List of Strings; traits to set for a node, or None. Mutually exclusive with trait. If not None, replaces the nodes traits with this list.

class ironic.api.controllers.v1.node.**NodeVIFController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

delete(*vif_id*)
Detach a VIF from this node

Parameters **vif_id** The ID of a VIF to detach

get_all()
Get a list of attached VIFs

post(*vif*)
Attach a VIF to this node

Parameters **vif** a dictionary of information about a VIF. It must have an id key, whose value is a unique identifier for that VIF.

class ironic.api.controllers.v1.node.**NodeVendorPassthruController**(**args*, ***kwargs*)

Bases: pecan.rest.RestController

REST controller for VendorPassthru.

This controller allow vendors to expose a custom functionality in the Ironic API. Ironic will merely relay the message from here to the appropriate driver, no introspection will be made in the message body.

methods(*node_ident*)

Retrieve information about vendor methods of the given node.

Parameters **node_ident** UUID or logical name of a node.

Returns dictionary with <vendor method name>:<method metadata> entries.

Raises NodeNotFound if the node is not found.

class ironic.api.controllers.v1.node.**NodesController**(**args*, ***kwargs*)

Bases: pecan.rest.RestController

REST controller for Nodes.

delete(*node_ident*, **args*)

Delete a node.

Parameters **node_ident** UUID or logical name of a node.

detail(*chassis_uuid=None*, *instance_uuid=None*, *associated=None*, *maintenance=None*, *retired=None*, *provision_state=None*, *marker=None*, *limit=None*, *sort_key='id'*, *sort_dir='asc'*, *driver=None*, *resource_class=None*, *fault=None*, *conductor_group=None*, *conductor=None*, *owner=None*, *description_contains=None*, *lessee=None*, *project=None*)

Retrieve a list of nodes with detail.

Parameters

- **chassis_uuid** Optional UUID of a chassis, to get only nodes for that chassis.
- **instance_uuid** Optional UUID of an instance, to find the node associated with that instance.
- **associated** Optional boolean whether to return a list of associated or unassociated nodes. May be combined with other parameters.
- **maintenance** Optional boolean value that indicates whether to get nodes in maintenance mode (True), or not in maintenance mode (False).
- **retired** Optional boolean value that indicates whether to get nodes which are retired.
- **provision_state** Optional string value to get only nodes in that provision state.
- **marker** pagination marker for large data sets.
- **limit** maximum number of resources to return in a single result. This value cannot be larger than the value of max_limit in the [api] section of the ironic configuration, or only max_limit resources will be returned.

- **sort_key** column to sort results by. Default: id.
- **sort_dir** direction to sort. asc or desc. Default: asc.
- **driver** Optional string value to get only nodes using that driver.
- **resource_class** Optional string value to get only nodes with that resource_class.
- **fault** Optional string value to get only nodes with that fault.
- **conductor_group** Optional string value to get only nodes with that conductor_group.
- **owner** Optional string value that set the owner whose nodes are to be retrurned.
- **lessee** Optional string value that set the lessee whose nodes are to be returned.
- **project** Optional string value that set the project - lessee or owner - whose nodes are to be returned.
- **description_contains** Optional string value to get only nodes with description field contains matching value.

from_chassis = False

A flag to indicate if the requests to this controller are coming from the top-level resource Chassis

get_all(*chassis_uuid=None*, *instance_uuid=None*, *associated=None*, *maintenance=None*, *retired=None*, *provision_state=None*, *marker=None*, *limit=None*, *sort_key='id'*, *sort_dir='asc'*, *driver=None*, *fields=None*, *resource_class=None*, *fault=None*, *conductor_group=None*, *detail=None*, *conductor=None*, *owner=None*, *description_contains=None*, *lessee=None*, *project=None*)

Retrieve a list of nodes.

Parameters

- **chassis_uuid** Optional UUID of a chassis, to get only nodes for that chassis.
- **instance_uuid** Optional UUID of an instance, to find the node associated with that instance.
- **associated** Optional boolean whether to return a list of associated or unassociated nodes. May be combined with other parameters.
- **maintenance** Optional boolean value that indicates whether to get nodes in maintenance mode (True), or not in maintenance mode (False).
- **retired** Optional boolean value that indicates whether to get retired nodes.
- **provision_state** Optional string value to get only nodes in that provision state.
- **marker** pagination marker for large data sets.
- **limit** maximum number of resources to return in a single result. This value cannot be larger than the value of max_limit in the [api] section of the ironic configuration, or only max_limit resources will be returned.

- **sort_key** column to sort results by. Default: id.
- **sort_dir** direction to sort. asc or desc. Default: asc.
- **driver** Optional string value to get only nodes using that driver.
- **resource_class** Optional string value to get only nodes with that resource_class.
- **conductor_group** Optional string value to get only nodes with that conductor_group.
- **conductor** Optional string value to get only nodes managed by that conductor.
- **owner** Optional string value that set the owner whose nodes are to be retrurned.
- **lessee** Optional string value that set the lessee whose nodes are to be returned.
- **project** Optional string value that set the project - lessee or owner - whose nodes are to be returned.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.
- **fault** Optional string value to get only nodes with that fault.
- **description_contains** Optional string value to get only nodes with description field contains matching value.

get_one(*node_ident*, *fields=None*)

Retrieve information about the given node.

Parameters

- **node_ident** UUID or logical name of a node.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.

invalid_sort_key_list = ['properties', 'driver_info', 'extra', 'instance_info', 'driver_internal_info', 'clean_step', 'deploy_step', 'raid_config', 'target_raid_config', 'traits', 'network_data']

maintenance = <ironic.api.controllers.v1.node.NodeMaintenanceController object>

Expose maintenance as a sub-element of nodes

management = <ironic.api.controllers.v1.node.NodeManagementController object>

Expose management as a sub-element of nodes

patch(*node_ident*, *reset_interfaces=None*, *patch=None*)

Update an existing node.

Parameters

- **node_ident** UUID or logical name of a node.

- **reset_interfaces** whether to reset hardware interfaces to their defaults. Only valid when updating the driver field.
- **patch** a json PATCH document to apply to this node.

post(*node*)

Create a new node.

Parameters **node** a node within the request body.

Example Node creation request:

```
{
    "name": "test_node_dynamic",
    "driver": "ipmi",
    "driver_info": {
        "ipmi_username": "ADMIN",
        "ipmi_password": "password"
    },
    "power_interface": "ipmitool",
    "resource_class": "bm-large"
}
```

states = <ironic.api.controllers.v1.node.NodeStatesController object>

Expose the state controller action as a sub-element of nodes

validate(*node=None*, *node_uuid=None*)

Validate the driver interfaces, using the nodes UUID or name.

Note that the node_uuid interface is deprecated in favour of the node interface

Parameters

- **node** UUID or name of a node.
- **node_uuid** UUID of a node.

vendor_passthru = <ironic.api.controllers.v1.node.NodeVendorPassthruController object>

A resource used for vendors to expose a custom functionality in the API

ironic.api.controllers.v1.node.**get_nodes_controller_reserved_names**()

ironic.api.controllers.v1.node.**hide_fields_in_newer_versions**(*obj*)

This method hides fields that were added in newer API versions.

Certain node fields were introduced at certain API versions. These fields are only made available when the requests API version matches or exceeds the versions when these fields were introduced.

ironic.api.controllers.v1.node.**indicator_convert_with_links**(*node_uuid*, *rpc_component*, *rpc_name*, ***rpc_fields*)

Add links to the indicator.

ironic.api.controllers.v1.node.**indicator_list_from_dict**(*node_ident*, *indicators*)

ironic.api.controllers.v1.node.**network_data_schema**()

ironic.api.controllers.v1.node.**node_convert_with_links**(*rpc_node*, *fields=None*, *sanitize=True*)

ironic.api.controllers.v1.node.**node_list_convert_with_links**(*nodes*, *limit*, *url*, *fields=None*, ***kwargs*)

ironic.api.controllers.v1.node.**node_patch_schema**()

ironic.api.controllers.v1.node.**node_patch_validator**(*name*, *value*)

ironic.api.controllers.v1.node.**node_sanitize**(*node*, *fields*, *cdict=None*, *show_driver_secrets=None*, *show_instance_secrets=None*, *evaluate_additional_policies=None*)

Removes sensitive and unrequested data.

Will only keep the fields specified in the `fields` parameter.

Parameters

- **fields** (*list of str*) list of fields to preserve, or `None` to preserve them all
- **cdict** Context dictionary for policy values evaluation. If not provided, it will be executed by the method, however for enumerating node lists, it is more efficent to provide.
- **show_driver_secrets** A boolean value to allow external single evaluation of policy instead of once per node. Default None.
- **show_instance_secrets** A boolean value to allow external evaluation of policy instead of once per node. Default None.
- **evaluate_additional_policies** A boolean value to allow external evaluation of policy instead of once per node. Default None.

ironic.api.controllers.v1.node.**node_schema**()

ironic.api.controllers.v1.node.**node_states_convert**(*rpc_node*)

ironic.api.controllers.v1.node.**node_validator**(*name*, *value*)

ironic.api.controllers.v1.node.**reject_fields_in_newer_versions**(*obj*)

When creating an object, reject fields that appear in newer versions.

ironic.api.controllers.v1.node.**reject_patch_in_newer_versions**(*patch*)

ironic.api.controllers.v1.node.**update_state_in_older_versions**(*obj*)

Change provision state names for API backwards compatibility.

Parameters **obj** The dict being returned to the API client that is to be updated by this method.

ironic.api.controllers.v1.node.**validate_network_data**(*network_data*)

Validates node network_data field.

This method validates network data configuration against JSON schema.

Parameters **network_data** a network_data field to validate

Raises Invalid if network data is not schema-compliant

ironic.api.controllers.v1.notification_utils module

ironic.api.controllers.v1.notification_utils.**emit_end_notification**(*context*, *obj*, *action*, ***kwargs*)

Helper for emitting API end notifications.

Parameters

- **context** request context.
- **obj** resource rpc object.
- **action** Action string to go in the EventType.
- **kwargs** kwargs to use when creating the notification payload.

ironic.api.controllers.v1.notification_utils.**emit_start_notification**(*context*, *obj*, *action*, ***kwargs*)

Helper for emitting API start notifications.

Parameters

- **context** request context.
- **obj** resource rpc object.
- **action** Action string to go in the EventType.
- **kwargs** kwargs to use when creating the notification payload.

ironic.api.controllers.v1.notification_utils.**handle_error_notification**(*context*, *obj*, *action*, ***kwargs*)

Context manager to handle any error notifications.

Parameters

- **context** request context.
- **obj** resource rpc object.
- **action** Action string to go in the EventType.
- **kwargs** kwargs to use when creating the notification payload.

ironic.api.controllers.v1.port module

class ironic.api.controllers.v1.port.**PortsController**(**args*, ***kwargs*)

Bases: pecan.rest.RestController

REST controller for Ports.

advanced_net_fields = ['pxe_enabled', 'local_link_connection']

delete(*port_uuid*)

Delete a port.

Parameters **port_uuid** UUID of a port.

Raises OperationNotPermitted, HTTPNotFound

detail(*node=None*, *node_uuid=None*, *address=None*, *marker=None*, *limit=None*, *sort_key='id'*, *sort_dir='asc'*, *portgroup=None*)

Retrieve a list of ports with detail.

Note that the node_uuid interface is deprecated in favour of the node interface

Parameters

- **node** UUID or name of a node, to get only ports for that node.
- **node_uuid** UUID of a node, to get only ports for that node.
- **address** MAC address of a port, to get the port which has this MAC address.
- **portgroup** UUID or name of a portgroup, to get only ports for that portgroup.
- **marker** pagination marker for large data sets.
- **limit** maximum number of resources to return in a single result. This value cannot be larger than the value of max_limit in the [api] section of the ironic configuration, or only max_limit resources will be returned.
- **sort_key** column to sort results by. Default: id.
- **sort_dir** direction to sort. asc or desc. Default: asc.

Raises NotAcceptable, HTTPNotFound

get_all(*node=None*, *node_uuid=None*, *address=None*, *marker=None*, *limit=None*, *sort_key='id'*, *sort_dir='asc'*, *fields=None*, *portgroup=None*, *detail=None*)

Retrieve a list of ports.

Note that the node_uuid interface is deprecated in favour of the node interface

Parameters

- **node** UUID or name of a node, to get only ports for that node.
- **node_uuid** UUID of a node, to get only ports for that node.
- **address** MAC address of a port, to get the port which has this MAC address.
- **marker** pagination marker for large data sets.
- **limit** maximum number of resources to return in a single result. This value cannot be larger than the value of max_limit in the [api] section of the ironic configuration, or only max_limit resources will be returned.
- **sort_key** column to sort results by. Default: id.
- **sort_dir** direction to sort. asc or desc. Default: asc.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.
- **portgroup** UUID or name of a portgroup, to get only ports for that portgroup.

Raises NotAcceptable, HTTPNotFound

get_one(*port_uuid*, *fields=None*)
Retrieve information about the given port.

Parameters

- **port_uuid** UUID of a port.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.

Raises NotAcceptable, HTTPNotFound

invalid_sort_key_list = ['extra', 'internal_info', 'local_link_connection']

patch(*port_uuid*, *patch*)
Update an existing port.

Parameters

- **port_uuid** UUID of a port.
- **patch** a json PATCH document to apply to this port.

Raises NotAcceptable, HTTPNotFound

post(*port*)
Create a new port.

Parameters **port** a port within the request body.

Raises NotAcceptable, HTTPNotFound, Conflict

ironic.api.controllers.v1.port.**convert_with_links**(*rpc_port*, *fields=None*, *sanitize=True*)

ironic.api.controllers.v1.port.**hide_fields_in_newer_versions**(*port*)

ironic.api.controllers.v1.port.**list_convert_with_links**(*rpc_ports*, *limit*, *url*, *fields=None*, ***kwargs*)

ironic.api.controllers.v1.port.**port_sanitize**(*port*, *fields=None*)
Removes sensitive and unrequested data.

Will only keep the fields specified in the `fields` parameter.

Parameters **fields** (*list of str*) list of fields to preserve, or `None` to preserve them all

ironic.api.controllers.v1.portgroup module

class ironic.api.controllers.v1.portgroup.**PortgroupsController**(**args*, ***kwargs*)
Bases: `pecan.rest.RestController`

REST controller for portgroups.

delete(*portgroup_ident*)
Delete a portgroup.

Parameters **portgroup_ident** UUID or logical name of a portgroup.

detail(*node=None*, *address=None*, *marker=None*, *limit=None*, *sort_key='id'*, *sort_dir='asc'*)
Retrieve a list of portgroups with detail.

Parameters

- **node** UUID or name of a node, to get only portgroups for that node.
- **address** MAC address of a portgroup, to get the portgroup which has this MAC address.
- **marker** pagination marker for large data sets.
- **limit** maximum number of resources to return in a single result. This value cannot be larger than the value of max_limit in the [api] section of the ironic configuration, or only max_limit resources will be returned.
- **sort_key** column to sort results by. Default: id.
- **sort_dir** direction to sort. asc or desc. Default: asc.

get_all(*node=None*, *address=None*, *marker=None*, *limit=None*, *sort_key='id'*, *sort_dir='asc'*, *fields=None*, *detail=None*)
Retrieve a list of portgroups.

Parameters

- **node** UUID or name of a node, to get only portgroups for that node.
- **address** MAC address of a portgroup, to get the portgroup which has this MAC address.
- **marker** pagination marker for large data sets.
- **limit** maximum number of resources to return in a single result. This value cannot be larger than the value of max_limit in the [api] section of the ironic configuration, or only max_limit resources will be returned.
- **sort_key** column to sort results by. Default: id.
- **sort_dir** direction to sort. asc or desc. Default: asc.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.

get_one(*portgroup_ident*, *fields=None*)
Retrieve information about the given portgroup.

Parameters

- **portgroup_ident** UUID or logical name of a portgroup.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.

invalid_sort_key_list = ['extra', 'internal_info', 'properties']

patch(*portgroup_ident*, *patch*)
Update an existing portgroup.

Parameters

- **portgroup_ident** UUID or logical name of a portgroup.
- **patch** a json PATCH document to apply to this portgroup.

post(*portgroup*)
Create a new portgroup.

Parameters **portgroup** a portgroup within the request body.

ironic.api.controllers.v1.portgroup.**convert_with_links**(*rpc_portgroup*, *fields=None*, *sanitize=True*)

Add links to the portgroup.

ironic.api.controllers.v1.portgroup.**list_convert_with_links**(*rpc_portgroups*, *limit*, *url*, *fields=None*, ***kwargs*)

ironic.api.controllers.v1.ramdisk module

class ironic.api.controllers.v1.ramdisk.**HeartbeatController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

Controller handling heartbeats from deploy ramdisk.

post(*node_ident*, *callback_url*, *agent_version=None*, *agent_token=None*, *agent_verify_ca=None*, *agent_status=None*, *agent_status_message=None*)
Process a heartbeat from the deploy ramdisk.

Parameters

- **node_ident** the UUID or logical name of a node.
- **callback_url** the URL to reach back to the ramdisk.
- **agent_version** The version of the agent that is heartbeating. None indicates that the agent that is heartbeating is a version before sending agent_version was introduced so agent v3.0.0 (the last release before sending agent_version was introduced) will be assumed.
- **agent_token** randomly generated validation token.
- **agent_verify_ca** TLS certificate to use to connect to the agent.
- **agent_status** Current status of the heartbeating agent. Used by anaconda ramdisk to send status back to Ironic. The valid states are start, end, error
- **agent_status_message** Optional status message describing current agent_status

Raises NodeNotFound if node with provided UUID or name was not found.

Raises InvalidUuidOrName if node_ident is not valid name or UUID.

Raises NoValidHost if RPC topic for node could not be retrieved.

Raises NotFound if requested API version does not allow this endpoint.

class ironic.api.controllers.v1.ramdisk.**LookupController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

Controller handling node lookup for a deploy ramdisk.

get_all(*addresses=None*, *node_uuid=None*)
Look up a node by its MAC addresses and optionally UUID.

If the restrict_lookup option is set to True (the default), limit the search to nodes in certain transient states (e.g. deploy wait).

Parameters

- **addresses** list of MAC addresses for a node.
- **node_uuid** UUID of a node.

Raises NotFound if requested API version does not allow this endpoint.

Raises NotFound if suitable node was not found or nodes provision state is not allowed for the lookup.

Raises IncompleteLookup if neither node UUID nor any valid MAC address was provided.

lookup_allowed(*node*)

ironic.api.controllers.v1.ramdisk.**config**(*token*)

ironic.api.controllers.v1.ramdisk.**convert_with_links**(*node*)

ironic.api.controllers.v1.utils module

class ironic.api.controllers.v1.utils.**PassthruResponse**(*obj*, *status_code=None*)

Bases: object

Object to hold the response from a passthru call

obj

Store the result object from the view

status_code

Store an optional status_code

ironic.api.controllers.v1.utils.**allow_agent_token**()

Check if agent token is available.

ironic.api.controllers.v1.utils.**allow_agent_version_in_heartbeat**()

Check if agent version is allowed to be passed into heartbeat.

Version 1.36 of the API added the ability for agents to pass their version information to Ironic on heartbeat.

ironic.api.controllers.v1.utils.**allow_allocation_backfill**()

Check if backfilling allocations is allowed.

Version 1.58 of the API added support for backfilling allocations.

ironic.api.controllers.v1.utils.**allow_allocation_owner**()

Check if allocation owner field is allowed.

Version 1.60 of the API added the owner field to the allocation object.

ironic.api.controllers.v1.utils.**allow_allocation_update**()

Check if updating an existing allocation is allowed or not.

Version 1.57 of the API added support for updating an allocation.

ironic.api.controllers.v1.utils.**allow_allocations**()
Check if accessing allocation endpoints is allowed.

Version 1.52 of the API exposed allocation endpoints and allocation_uuid field for the node.

ironic.api.controllers.v1.utils.**allow_bios_interface**()
Check if we should support bios interface and endpoints.

Version 1.40 of the API added support for bios interface.

ironic.api.controllers.v1.utils.**allow_build_configdrive**()
Check if building configdrive is allowed.

Version 1.56 of the API added support for building configdrive.

ironic.api.controllers.v1.utils.**allow_configdrive_vendor_data**()
Check if configdrive can contain a vendor_data key.

Version 1.59 of the API added support for configdrive vendor_data.

ironic.api.controllers.v1.utils.**allow_deploy_steps**()
Check if deploy_steps are available.

ironic.api.controllers.v1.utils.**allow_deploy_templates**()
Check if accessing deploy template endpoints is allowed.

Version 1.55 of the API exposed deploy template endpoints.

ironic.api.controllers.v1.utils.**allow_detail_query**()
Check if passing a detail=True query string is allowed.

Version 1.43 allows a user to pass the detail query string to list the resource with all the fields.

ironic.api.controllers.v1.utils.**allow_dynamic_drivers**()
Check if dynamic driver API calls are allowed.

Version 1.30 of the API added support for all of the driver composition related calls in the /v1/drivers API.

ironic.api.controllers.v1.utils.**allow_dynamic_interfaces**()
Check if dynamic interface fields are allowed.

Version 1.31 of the API added support for viewing and setting the fields in `V31_FIELDS` on the node object.

ironic.api.controllers.v1.utils.**allow_expose_conductors**()
Check if accessing conductor endpoints is allowed.

Version 1.49 of the API exposed conductor endpoints and conductor field for the node.

ironic.api.controllers.v1.utils.**allow_expose_events**()
Check if accessing events endpoint is allowed.

Version 1.54 of the API added the events endpoint.

ironic.api.controllers.v1.utils.**allow_field**(*field*)
Check if a field is allowed in the current version.

ironic.api.controllers.v1.utils.**allow_inject_nmi**()
Check if Inject NMI is allowed for the node.

Version 1.29 of the API allows Inject NMI for the node.

ironic.api.controllers.v1.utils.**allow_inspect_abort**()
Check if inspection abort is allowed.

Version 1.41 of the API added support for inspection abort

ironic.api.controllers.v1.utils.**allow_inspect_wait_state**()
Check if inspect wait is allowed for the node.

Version 1.39 of the API adds inspect wait state to substitute inspecting state during asynchronous hardware inspection.

ironic.api.controllers.v1.utils.**allow_links_node_states_and_driver_properties**()
Check if links are displayable.

Version 1.14 of the API allows the display of links to node states and driver properties.

ironic.api.controllers.v1.utils.**allow_local_link_connection_network_type**()
Check if network_type is allowed in ports link_local_connection

ironic.api.controllers.v1.utils.**allow_node_history**()
Check if node history access is permitted by API version.

ironic.api.controllers.v1.utils.**allow_node_logical_names**()

ironic.api.controllers.v1.utils.**allow_node_rebuild_with_configdrive**()
Check if we should support node rebuild with configdrive.

Version 1.35 of the API added support for node rebuild with configdrive.

ironic.api.controllers.v1.utils.**allow_port_advanced_net_fields**()
Check if we should return local_link_connection and pxe_enabled fields.

Version 1.19 of the API added support for these new fields in port object.

ironic.api.controllers.v1.utils.**allow_port_internal_info**()
Check if accessing internal_info is allowed for the port.

Version 1.18 of the API exposes internal_info readonly field for the port.

ironic.api.controllers.v1.utils.**allow_port_is_smartnic**()
Check if port is_smartnic field is allowed.

Version 1.53 of the API added is_smartnic field to the port object.

ironic.api.controllers.v1.utils.**allow_port_physical_network**()
Check if port physical network field is allowed.

Version 1.34 of the API added the physical network field to the port object. We also check whether the target version of the Port object supports the physical_network field as this may not be the case during a rolling upgrade.

ironic.api.controllers.v1.utils.**allow_portgroup_mode_properties**()
Check if mode and properties can be added to/queried from a portgroup.

Version 1.26 of the API added mode and properties fields to portgroup object.

ironic.api.controllers.v1.utils.**allow_portgroups**()
Check if we should support portgroup operations.

Version 1.23 of the API added support for PortGroups.

ironic.api.controllers.v1.utils.**allow_portgroups_subcontrollers**()
Check if portgroups can be used as subcontrollers.

Version 1.24 of the API added support for Portgroups as subcontrollers

ironic.api.controllers.v1.utils.**allow_query_bios**()
Check if BIOS queries should be allowed based on version

ironic.api.controllers.v1.utils.**allow_raid_config**()
Check if RAID configuration is allowed for the node.

Version 1.12 of the API allows RAID configuration for the node.

ironic.api.controllers.v1.utils.**allow_ramdisk_endpoints**()
Check if heartbeat and lookup endpoints are allowed.

Version 1.22 of the API introduced them.

ironic.api.controllers.v1.utils.**allow_remove_chassis_uuid**()
Check if chassis_uuid can be removed from node.

Version 1.25 of the API added support for chassis_uuid removal

ironic.api.controllers.v1.utils.**allow_rescue_interface**()
Check if we should support rescue and unrescue operations and interface.

Version 1.38 of the API added support for rescue and unrescue.

ironic.api.controllers.v1.utils.**allow_reset_interfaces**()
Check if passing a reset_interfaces query string is allowed.

ironic.api.controllers.v1.utils.**allow_soft_power_off**()
Check if Soft Power Off is allowed for the node.

Version 1.27 of the API allows Soft Power Off, including Soft Reboot, for the node.

ironic.api.controllers.v1.utils.**allow_status_in_heartbeat**()
Check if heartbeat accepts agent_status and agent_status_message.

ironic.api.controllers.v1.utils.**allow_storage_interface**()
Check if we should support storage_interface node and driver fields.

Version 1.33 of the API added support for storage interfaces.

ironic.api.controllers.v1.utils.**allow_traits**()
Check if traits are allowed for the node.

Version 1.37 of the API allows traits for the node.

ironic.api.controllers.v1.utils.**allow_verify_ca_in_heartbeat**()
Check if heartbeat accepts agent_verify_ca.

ironic.api.controllers.v1.utils.**allow_vifs_subcontroller**()
Check if node/vifs can be used.

Version 1.28 of the API added support for VIFs to be attached to Nodes.

ironic.api.controllers.v1.utils.**allow_volume**()
Check if volume connectors and targets are allowed.

Version 1.32 of the API added support for volume connectors and targets

ironic.api.controllers.v1.utils.**apply_jsonpatch**(*doc*, *patch*)
Apply a JSON patch, one operation at a time.

If the patch fails to apply, this allows us to determine which operation failed, making the error message a little less cryptic.

Parameters

- **doc** The JSON document to patch.
- **patch** The JSON patch to apply.

Returns The result of the patch operation.

Raises PatchError if the patch fails to apply.

Raises exception.ClientSideError if the patch adds a new root attribute.

ironic.api.controllers.v1.utils.**check_allocation_policy_and_retrieve**(*policy_name*, *allocation_ident*)
Check if the specified policy authorizes request on allocation.

Param policy_name: Name of the policy to check.

Param allocation_ident: the UUID or logical name of a node.

Raises HTTPForbidden if the policy forbids access.

Raises AllocationNotFound if the node is not found.

Returns RPC node identified by node_ident

ironic.api.controllers.v1.utils.**check_allow_clean_disable_ramdisk**(*target*, *disable_ramdisk*)

ironic.api.controllers.v1.utils.**check_allow_configdrive**(*target*, *configdrive=None*)

ironic.api.controllers.v1.utils.**check_allow_deploy_steps**(*target*, *deploy_steps*)
Check if deploy steps are allowed

ironic.api.controllers.v1.utils.**check_allow_driver_detail**(*detail*)
Check if getting detailed driver info is allowed.

Version 1.30 of the API allows this.

ironic.api.controllers.v1.utils.**check_allow_filter_by_conductor**(*conductor*)
Check if filtering nodes by conductor is allowed.

Version 1.49 of the API allows filtering nodes by conductor.

ironic.api.controllers.v1.utils.**check_allow_filter_by_conductor_group**(*conductor_group*)
Check if filtering nodes by conductor_group is allowed.

Version 1.46 of the API allows filtering nodes by conductor_group.

ironic.api.controllers.v1.utils.**check_allow_filter_by_fault**(*fault*)
Check if filtering nodes by fault is allowed.

Version 1.42 of the API allows filtering nodes by fault.

ironic.api.controllers.v1.utils.**check_allow_filter_by_lessee**(*lessee*)
Check if filtering nodes by lessee is allowed.

Version 1.62 of the API allows filtering nodes by lessee.

ironic.api.controllers.v1.utils.**check_allow_filter_by_owner**(*owner*)
Check if filtering nodes by owner is allowed.

Version 1.50 of the API allows filtering nodes by owner.

ironic.api.controllers.v1.utils.**check_allow_filter_driver_type**(*driver_type*)
Check if filtering drivers by classic/dynamic is allowed.

Version 1.30 of the API allows this.

ironic.api.controllers.v1.utils.**check_allow_management_verbs**(*verb*)

ironic.api.controllers.v1.utils.**check_allow_specify_driver**(*driver*)
Check if filtering nodes by driver is allowed.

Version 1.16 of the API allows filter nodes by driver.

ironic.api.controllers.v1.utils.**check_allow_specify_fields**(*fields*)
Check if fetching a subset of the resource attributes is allowed.

Version 1.8 of the API allows fetching a subset of the resource attributes, this method checks if the required version is being requested.

ironic.api.controllers.v1.utils.**check_allow_specify_resource_class**(*resource_class*)
Check if filtering nodes by resource_class is allowed.

Version 1.21 of the API allows filtering nodes by resource_class.

ironic.api.controllers.v1.utils.**check_allowed_fields**(*fields*)
Check if fetching a particular field is allowed.

This method checks if the required version is being requested for fields that are only allowed to be fetched in a particular API version.

ironic.api.controllers.v1.utils.**check_allowed_portgroup_fields**(*fields*)
Check if fetching a particular field of a portgroup is allowed.

This method checks if the required version is being requested for fields that are only allowed to be fetched in a particular API version.

ironic.api.controllers.v1.utils.**check_for_invalid_fields**(*fields*, *object_fields*)
Check for requested non-existent fields.

Check if the user requested non-existent fields.

Parameters **fields** A list of fields requested by the user

Object_fields A list of fields supported by the object.

Raises InvalidParameterValue if invalid fields were requested.

ironic.api.controllers.v1.utils.**check_for_invalid_state_and_allow_filter**(*provision_state*)
Check if filtering nodes by provision state is allowed.

Version 1.9 of the API allows filter nodes by provision state.

ironic.api.controllers.v1.utils.**check_list_policy**(*object_type*, *owner=None*)
Check if the list policy authorizes this request on an object.

Param object_type: type of object being checked

Param owner: owner filter for list query, if any

Raises HTTPForbidden if the policy forbids access.

Returns owner that should be used for list query, if needed

ironic.api.controllers.v1.utils.**check_multiple_node_policies_and_retrieve**(*policy_names*, *node_ident*, *with_suffix=False*)

Check if the specified policies authorize this request on a node.

Param policy_names: List of policy names to check.

Param node_ident: the UUID or logical name of a node.

Param with_suffix: whether the RPC node should include the suffix

Raises HTTPForbidden if the policy forbids access.

Raises NodeNotFound if the node is not found.

Returns RPC node identified by node_ident

ironic.api.controllers.v1.utils.**check_node_policy_and_retrieve**(*policy_name*, *node_ident*, *with_suffix=False*)

Check if the specified policy authorizes this request on a node.

Param policy_name: Name of the policy to check.

Param node_ident: the UUID or logical name of a node.

Param with_suffix: whether the RPC node should include the suffix

Raises HTTPForbidden if the policy forbids access.

Raises NodeNotFound if the node is not found.

Returns RPC node identified by node_ident

ironic.api.controllers.v1.utils.**check_owner_policy**(*object_type*, *policy_name*, *owner*, *lessee=None*, *conceal_node=False*)

Check if the policy authorizes this request on an object.

Param object_type: type of object being checked

Param policy_name: Name of the policy to check.

Param owner: the owner

Param lessee: the lessee

Param conceal_node: the UUID of the node IF we should conceal the existence of the node with a 404 Error instead of a 403 Error.

Raises HTTPForbidden if the policy forbids access.

ironic.api.controllers.v1.utils.**check_policy**(*policy_name*)

Check if the specified policy is authorised for this request.

Policy_name Name of the policy to check.

Raises HTTPForbidden if the policy forbids access.

ironic.api.controllers.v1.utils.**check_policy_true**(*policy_name*)

Check if the specified policy is authorised for this request.

Policy_name Name of the policy to check.

Returns True if policy is matched, otherwise false.

ironic.api.controllers.v1.utils.**check_port_list_policy**(*portgroup=False*, *parent_node=None*, *parent_portgroup=None*)

Check if the specified policy authorizes this request on a port.

Parameters

- **portgroup** Boolean value, default false, indicating if the list policy check is for a portgroup as the policy names are different between ports and portgroups.
- **parent_node** The UUID of a node, if any, to apply a policy check to as well before applying other policy check operations.
- **parent_portgroup** The UUID of the parent portgroup if the list of ports was retrieved via the /v1/portgroups/<uuid>/ports.

Raises HTTPForbidden if the policy forbids access.

Returns owner that should be used for list query, if needed

ironic.api.controllers.v1.utils.**check_port_policy_and_retrieve**(*policy_name*, *port_ident*, *portgroup=False*)

Check if the specified policy authorizes this request on a port.

Param policy_name: Name of the policy to check.

Param port_ident: The name, uuid, or other valid ID value to find a port or portgroup by.

Raises HTTPForbidden if the policy forbids access.

Raises NodeNotFound if the node is not found.

Returns RPC port identified by port_ident associated node

ironic.api.controllers.v1.utils.**check_volume_list_policy**(*parent_node=None*)

Check if the specified policy authorizes this request on a volume.

Parameters **parent_node** The UUID of a node, if any, to apply a policy check to as well before applying other policy check operations.

Raises HTTPForbidden if the policy forbids access.

Returns owner that should be used for list query, if needed

ironic.api.controllers.v1.utils.**check_volume_policy_and_retrieve**(*policy_name*, *vol_ident*, *target=False*)

Check if the specified policy authorizes this request on a volume.

Param policy_name: Name of the policy to check.

Param vol_ident: The name, uuid, or other valid ID value to find a volume target or connector by.

Param target: Boolean value to indicate if the check is for a volume target or connector. Default value is False, implying connector.

Raises HTTPForbidden if the policy forbids access.

Raises VolumeConnectorNotFound if the node is not found.

Raises VolumeTargetNotFound if the node is not found.

Returns RPC port identified by port_ident associated node

ironic.api.controllers.v1.utils.**disallowed_fields**()
Generator of fields not allowed in the current request.

ironic.api.controllers.v1.utils.**get_controller_reserved_names**(*cls*)
Get reserved names for a given controller.

Inspect the controller class and return the reserved names within it. Reserved names are names that can not be used as an identifier for a resource because the names are either being used as a custom action or is the name of a nested controller inside the given class.

Parameters **cls** The controller class to be inspected.

ironic.api.controllers.v1.utils.**get_patch_values**(*patch*, *path*)
Get the patch values corresponding to the specified path.

If there are multiple values specified for the same path, for example

```
[{'op': 'add', 'path': '/name', 'value': 'abc'},
 {'op': 'add', 'path': '/name', 'value': 'bca'}]
```

return all of them in a list (preserving order)

Parameters

- **patch** HTTP PATCH request body.
- **path** the path to get the patch values for.

Returns list of values for the specified path in the patch.

ironic.api.controllers.v1.utils.**get_request_return_fields**(*fields*, *detail*, *default_fields*, *check_detail_version=<function allow_detail_query>*, *check_fields_version=None*)
Calculate fields to return from an API request

The fields query and detail=True query can not be passed into a request at the same time. To use the detail query we need to be on a version of the API greater than expected, likewise some APIs require a certain version for the fields query. This function raises an InvalidParameterValue exception if any of these conditions are not met.

If these checks pass then this function will return either the fields passed in or the default fields provided.

Parameters

- **fields** The fields query passed into the API request.
- **detail** The detail query passed into the API request.
- **default_fields** The default fields to return if fields=None and detail=None.
- **check_detail_version** Function to check if detail query is allowed based on the version.

- **check_fields_version** Function to check if fields query is allowed based on the version.

Raises InvalidParameterValue if there is an invalid combination of query strings or API version.

Returns fields passed in value or default_fields

ironic.api.controllers.v1.utils.**get_rpc_allocation**(*allocation_ident*)
Get the RPC allocation from the allocation UUID or logical name.

Parameters **allocation_ident** the UUID or logical name of an allocation.

Returns The RPC allocation.

Raises InvalidUuidOrName if the name or uuid provided is not valid.

Raises AllocationNotFound if the allocation is not found.

ironic.api.controllers.v1.utils.**get_rpc_allocation_with_suffix**(*allocation_ident*)
Get the RPC allocation from the allocation UUID or logical name.

If HAS_JSON_SUFFIX flag is set in the pecan environment, try also looking for allocation_ident with .json suffix. Otherwise identical to get_rpc_allocation.

Parameters **allocation_ident** the UUID or logical name of an allocation.

Returns The RPC allocation.

Raises InvalidUuidOrName if the name or uuid provided is not valid.

Raises AllocationNotFound if the allocation is not found.

ironic.api.controllers.v1.utils.**get_rpc_deploy_template**(*template_ident*)
Get the RPC deploy template from the UUID or logical name.

Parameters **template_ident** the UUID or logical name of a deploy template.

Returns The RPC deploy template.

Raises InvalidUuidOrName if the name or uuid provided is not valid.

Raises DeployTemplateNotFound if the deploy template is not found.

ironic.api.controllers.v1.utils.**get_rpc_deploy_template_with_suffix**(*template_ident*)
Get the RPC deploy template from the UUID or logical name.

If HAS_JSON_SUFFIX flag is set in the pecan environment, try also looking for template_ident with .json suffix. Otherwise identical to get_rpc_deploy_template.

Parameters **template_ident** the UUID or logical name of a deploy template.

Returns The RPC deploy template.

Raises InvalidUuidOrName if the name or uuid provided is not valid.

Raises DeployTemplateNotFound if the deploy template is not found.

ironic.api.controllers.v1.utils.**get_rpc_node**(*node_ident*)
Get the RPC node from the node uuid or logical name.

Parameters **node_ident** the UUID or logical name of a node.

Returns The RPC Node.

Raises InvalidUuidOrName if the name or uuid provided is not valid.

Raises NodeNotFound if the node is not found.

ironic.api.controllers.v1.utils.**get_rpc_node_with_suffix**(*node_ident*)
Get the RPC node from the node uuid or logical name.

If HAS_JSON_SUFFIX flag is set in the pecan environment, try also looking for node_ident with .json suffix. Otherwise identical to get_rpc_node.

Parameters **node_ident** the UUID or logical name of a node.

Returns The RPC Node.

Raises InvalidUuidOrName if the name or uuid provided is not valid.

Raises NodeNotFound if the node is not found.

ironic.api.controllers.v1.utils.**get_rpc_portgroup**(*portgroup_ident*)
Get the RPC portgroup from the portgroup UUID or logical name.

Parameters **portgroup_ident** the UUID or logical name of a portgroup.

Returns The RPC portgroup.

Raises InvalidUuidOrName if the name or uuid provided is not valid.

Raises PortgroupNotFound if the portgroup is not found.

ironic.api.controllers.v1.utils.**get_rpc_portgroup_with_suffix**(*portgroup_ident*)
Get the RPC portgroup from the portgroup UUID or logical name.

If HAS_JSON_SUFFIX flag is set in the pecan environment, try also looking for portgroup_ident with .json suffix. Otherwise identical to get_rpc_portgroup.

Parameters **portgroup_ident** the UUID or logical name of a portgroup.

Returns The RPC portgroup.

Raises InvalidUuidOrName if the name or uuid provided is not valid.

Raises PortgroupNotFound if the portgroup is not found.

ironic.api.controllers.v1.utils.**initial_node_provision_state**()
Return node state to use by default when creating new nodes.

Previously the default state for new nodes was AVAILABLE. Starting with API 1.11 it is ENROLL.

ironic.api.controllers.v1.utils.**is_path_removed**(*patch*, *path*)
Returns whether the patch includes removal of the path (or subpath of).

Parameters

- **patch** HTTP PATCH request body.
- **path** the path to check.

Returns True if path or subpath being removed, False otherwise.

ironic.api.controllers.v1.utils.**is_path_updated**(*patch*, *path*)
Returns whether the patch includes operation on path (or its subpath).

Parameters

- **patch** HTTP PATCH request body.

- **path** the path to check.

Returns True if path or subpath being patched, False otherwise.

ironic.api.controllers.v1.utils.**is_valid_logical_name**(*name*)
Determine if the provided name is a valid hostname.

ironic.api.controllers.v1.utils.**is_valid_node_name**(*name*)
Determine if the provided name is a valid node name.

Check to see that the provided node name is valid, and isnt a UUID.

Parameters **name** the node name to check.

Returns True if the name is valid, False otherwise.

ironic.api.controllers.v1.utils.**local_link_normalize**(*name*, *value*)

ironic.api.controllers.v1.utils.**object_to_dict**(*obj*, *include_created_at=True*, *include_updated_at=True*, *include_uuid=True*, *link_resource=None*, *link_resource_args=None*, *fields=None*)
Helper function to convert RPC objects to REST API dicts.

Parameters

- **obj** RPC object to convert to a dict
- **include_created_at** Whether to include standard base class attribute created_at
- **include_updated_at** Whether to include standard base class attribute updated_at
- **include_uuid** Whether to include standard base class attribute uuid
- **link_resource** When specified, generate a `links` value with a `self` and `bookmark` using this resource name
- **link_resource_args** Resource arguments to be added to generated links. When not specified, the object `uuid` will be used.
- **fields** Key names for dict values to populate directly from object attributes

Returns A dict containing values from the object

ironic.api.controllers.v1.utils.**patch_update_changed_fields**(*from_dict*, *rpc_object*, *fields*, *schema*, *id_map=None*)
Update rpc object based on changed fields in a dict.

Only fields which have a corresponding schema field are updated when changed. Other values can be updated using the id_map.

Parameters

- **from_dict** Dict containing changed field values
- **rpc_object** Object to update changed fields on
- **fields** Field names on the rpc object
- **schema** jsonschema to get field names of the dict

- **id_map** Optional dict mapping object field names to arbitrary values when there is no matching field in the schema

ironic.api.controllers.v1.utils.**patch_validate_allowed_fields**(*patch*, *allowed_fields*)

Validate that a patch list only modifies allowed fields.

Parameters

- **patch** List of patch dicts to validate
- **allowed_fields** List of fields which are allowed to be patched

Returns The list of fields which will be patched

Raises exception.Invalid if any patch changes a field not in `allowed_fields`

ironic.api.controllers.v1.utils.**patched_validate_with_schema**(*patched_dict*, *schema*, *validator=None*)

Validate a patched dict object against a validator or schema.

This function has the side-effect of deleting any dict value which is not in the schema. This allows database-loaded objects to be pruned of their internal values before validation.

Parameters

- **patched_dict** dict representation of the object with patch updates applied
- **schema** Any dict key not in the schema will be deleted from the dict. If no validator is specified then the resulting `patched_dict` will be validated agains the schema
- **validator** Optional validator to use if there is extra validation required beyond the schema

Raises exception.Invalid if validation fails

ironic.api.controllers.v1.utils.**populate_node_uuid**(*obj*, *to_dict*)

Look up the node referenced in the object and populate a dict.

The node is fetched with the object `node_id` attribute and the dict `node_uuid` value is populated with the node uuid

Parameters

- **obj** object to get the node_id attribute
- **to_dict** dict to populate with a `node_uuid` value

Raises exception.NodeNotFound if the node is not found

ironic.api.controllers.v1.utils.**replace_node_id_with_uuid**(*to_dict*)

Replace `node_id` dict value with `node_uuid`

`node_uuid` is found by fetching the node by id lookup.

Parameters **to_dict** Dict to set `node_uuid` value on

Returns The node object from the lookup

Raises NodeNotFound with status_code set to 400 BAD_REQUEST when node is not found.

ironic.api.controllers.v1.utils.**replace_node_uuid_with_id**(*to_dict*)
Replace node_uuid dict value with node_id

node_id is found by fetching the node by uuid lookup.

Parameters **to_dict** Dict to set node_id value on

Returns The node object from the lookup

Raises NodeNotFound with status_code set to 400 BAD_REQUEST when node is not found.

ironic.api.controllers.v1.utils.**sanitize_dict**(*to_sanitize*, *fields*)
Removes sensitive and unrequested data.

Will only keep the fields specified in the fields parameter (plus the links field).

Parameters

- **to_sanitize** dict to sanitize
- **fields** (*list of str*) list of fields to preserve, or None to preserve them all

ironic.api.controllers.v1.utils.**validate_limit**(*limit*)

ironic.api.controllers.v1.utils.**validate_sort_dir**(*sort_dir*)

ironic.api.controllers.v1.utils.**vendor_passthru**(*ident*, *method*, *topic*, *data=None*, *driver_passthru=False*)
Call a vendor passthru API extension.

Call the vendor passthru API extension and process the method response to set the right return code for methods that are asynchronous or synchronous; Attach the return value to the response object if its being served statically.

Parameters

- **ident** The resource identification. For nodes vendor passthru this is the nodes UUID, for drivers vendor passthru this is the drivers name.
- **method** The vendor method name.
- **topic** The RPC topic.
- **data** The data passed to the vendor method. Defaults to None.
- **driver_passthru** Boolean value. Whether this is a node or driver vendor passthru. Defaults to False.

Returns A WSME response object to be returned by the API.

ironic.api.controllers.v1.versions module

ironic.api.controllers.v1.versions.**max_version_string**()
Returns the maximum supported API version (as a string).

If the service is pinned, the maximum API version is the pinned version. Otherwise, it is the maximum supported API version.

ironic.api.controllers.v1.versions.**min_version_string**()
Returns the minimum supported API version (as a string)

ironic.api.controllers.v1.volume module

class ironic.api.controllers.v1.volume.**VolumeController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

REST controller for volume root

get()

ironic.api.controllers.v1.volume.**convert**(*node_ident=None*)

ironic.api.controllers.v1.volume_connector module

class ironic.api.controllers.v1.volume_connector.**VolumeConnectorsController**(**args*, ***kwargs*)
Bases: pecan.rest.RestController

REST controller for VolumeConnectors.

delete(*connector_uuid*)
Delete a volume connector.

Parameters **connector_uuid** UUID of a volume connector.

Raises OperationNotPermitted if accessed with specifying a parent node.

Raises NodeLocked if node is locked by another conductor

Raises NodeNotFound if the node associated with the connector does not exist

Raises VolumeConnectorNotFound if the volume connector cannot be found

Raises InvalidStateRequested If a node associated with the volume connector is not powered off.

get_all(*node=None*, *marker=None*, *limit=None*, *sort_key='id'*, *sort_dir='asc'*, *fields=None*, *detail=None*, *project=None*)
Retrieve a list of volume connectors.

Parameters

- **node** UUID or name of a node, to get only volume connectors for that node.
- **marker** pagination marker for large data sets.

- **limit** maximum number of resources to return in a single result. This value cannot be larger than the value of max_limit in the [api] section of the ironic configuration, or only max_limit resources will be returned.
- **sort_key** column to sort results by. Default: id.
- **sort_dir** direction to sort. asc or desc. Default: asc.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.
- **detail** Optional, whether to retrieve with detail.

Returns a list of volume connectors, or an empty list if no volume connector is found.

Raises InvalidParameterValue if sort_key does not exist

Raises InvalidParameterValue if sort key is invalid for sorting.

Raises InvalidParameterValue if both fields and detail are specified.

get_one(*connector_uuid*, *fields=None*)

Retrieve information about the given volume connector.

Parameters

- **connector_uuid** UUID of a volume connector.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.

Returns API-serializable volume connector object.

Raises OperationNotPermitted if accessed with specifying a parent node.

Raises VolumeConnectorNotFound if no volume connector exists with the specified UUID.

invalid_sort_key_list = ['extra']

patch(*connector_uuid*, *patch*)

Update an existing volume connector.

Parameters

- **connector_uuid** UUID of a volume connector.
- **patch** a json PATCH document to apply to this volume connector.

Returns API-serializable volume connector object.

Raises OperationNotPermitted if accessed with specifying a parent node.

Raises PatchError if a given patch can not be applied.

Raises VolumeConnectorNotFound if no volume connector exists with the specified UUID.

Raises InvalidParameterValue if the volume connectors UUID is being changed

Raises NodeLocked if node is locked by another conductor

Raises NodeNotFound if the node associated with the connector does not exist

Raises VolumeConnectorTypeAndIdAlreadyExists if another connector already exists with the same values for type and connector_id fields

Raises InvalidUUID if invalid node UUID is passed in the patch.

Raises InvalidStateRequested If a node associated with the volume connector is not powered off.

post(*connector*)
Create a new volume connector.

Parameters **connector** a volume connector within the request body.

Returns API-serializable volume connector object.

Raises OperationNotPermitted if accessed with specifying a parent node.

Raises VolumeConnectorTypeAndIdAlreadyExists if a volume connector already exists with the same type and connector_id

Raises VolumeConnectorAlreadyExists if a volume connector with the same UUID already exists

ironic.api.controllers.v1.volume_connector.**convert_with_links**(*rpc_connector*, *fields=None*, *sanitize=True*)

ironic.api.controllers.v1.volume_connector.**list_convert_with_links**(*rpc_connectors*, *limit*, *url*, *fields=None*, *detail=None*, ***kwargs*)

ironic.api.controllers.v1.volume_target module

class ironic.api.controllers.v1.volume_target.**VolumeTargetsController**(**args*, ***kwargs*)

Bases: pecan.rest.RestController

REST controller for VolumeTargets.

delete(*target_uuid*)
Delete a volume target.

Parameters **target_uuid** UUID of a volume target.

Raises OperationNotPermitted if accessed with specifying a parent node.

Raises NodeLocked if node is locked by another conductor

Raises NodeNotFound if the node associated with the target does not exist

Raises VolumeTargetNotFound if the volume target cannot be found

Raises InvalidStateRequested If a node associated with the volume target is not powered off.

get_all(*node=None*, *marker=None*, *limit=None*, *sort_key='id'*, *sort_dir='asc'*, *fields=None*, *detail=None*, *project=None*)
Retrieve a list of volume targets.

Parameters

- **node** UUID or name of a node, to get only volume targets for that node.
- **marker** pagination marker for large data sets.
- **limit** maximum number of resources to return in a single result. This value cannot be larger than the value of max_limit in the [api] section of the ironic configuration, or only max_limit resources will be returned.
- **sort_key** column to sort results by. Default: id.
- **sort_dir** direction to sort. asc or desc. Default: asc.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.
- **detail** Optional, whether to retrieve with detail.
- **project** Optional, an associated node project (owner, or lessee) to filter the query upon.

Returns a list of volume targets, or an empty list if no volume target is found.

Raises InvalidParameterValue if sort_key does not exist

Raises InvalidParameterValue if sort key is invalid for sorting.

Raises InvalidParameterValue if both fields and detail are specified.

get_one(*target_uuid*, *fields=None*)

Retrieve information about the given volume target.

Parameters

- **target_uuid** UUID of a volume target.
- **fields** Optional, a list with a specified set of fields of the resource to be returned.

Returns API-serializable volume target object.

Raises OperationNotPermitted if accessed with specifying a parent node.

Raises VolumeTargetNotFound if no volume target with this UUID exists

invalid_sort_key_list = ['extra', 'properties']

patch(*target_uuid*, *patch*)

Update an existing volume target.

Parameters

- **target_uuid** UUID of a volume target.
- **patch** a json PATCH document to apply to this volume target.

Returns API-serializable volume target object.

Raises OperationNotPermitted if accessed with specifying a parent node.

Raises PatchError if a given patch can not be applied.

Raises InvalidParameterValue if the volume targets UUID is being changed

Raises NodeLocked if the node is already locked

Raises NodeNotFound if the node associated with the volume target does not exist

Raises VolumeTargetNotFound if the volume target cannot be found

Raises VolumeTargetBootIndexAlreadyExists if a volume target already exists with the same node ID and boot index values

Raises InvalidUUID if invalid node UUID is passed in the patch.

Raises InvalidStateRequested If a node associated with the volume target is not powered off.

post(*target*)

Create a new volume target.

Parameters **target** a volume target within the request body.

Returns API-serializable volume target object.

Raises OperationNotPermitted if accessed with specifying a parent node.

Raises VolumeTargetBootIndexAlreadyExists if a volume target already exists with the same node ID and boot index

Raises VolumeTargetAlreadyExists if a volume target with the same UUID exists

ironic.api.controllers.v1.volume_target.**convert_with_links**(*rpc_target*, *fields=None*, *sanitize=True*)

ironic.api.controllers.v1.volume_target.**list_convert_with_links**(*rpc_targets*, *limit*, *url*, *fields=None*, *detail=None*, ***kwargs*)

Module contents

Version 1 of the Ironic API

Specification can be found at doc/source/webapi/v1.rst

class ironic.api.controllers.v1.**Controller**

Bases: object

Version 1 API controller root.

index()

Submodules

ironic.api.controllers.base module

class ironic.api.controllers.base.**Version**(*headers*, *default_version*, *latest_version*)

Bases: object

API Version object.

max_string = 'X-OpenStack-Ironic-API-Maximum-Version'

HTTP response header

min_string = 'X-OpenStack-Ironic-API-Minimum-Version'
HTTP response header

static parse_headers(*headers*, *default_version*, *latest_version*)
Determine the API version requested based on the headers supplied.

Parameters

- **headers** webob headers
- **default_version** version to use if not specified in headers
- **latest_version** version to use if latest is requested

Returns a tuple of (major, minor) version numbers

Raises webob.HTTPNotAcceptable

string = 'X-OpenStack-Ironic-API-Version'
HTTP Header string carrying the requested version

ironic.api.controllers.link module

ironic.api.controllers.link.**build_url**(*resource*, *resource_args*, *bookmark=False*, *base_url=None*)

ironic.api.controllers.link.**make_link**(*rel_name*, *url*, *resource*, *resource_args*, *bookmark=False*, *type=None*)
Build a dict representing a link

ironic.api.controllers.root module

class ironic.api.controllers.root.**RootController**
Bases: object

index(**args*)

ironic.api.controllers.root.**root**()

ironic.api.controllers.version module

ironic.api.controllers.version.**all_versions**()

ironic.api.controllers.version.**default_version**()
Return a dict representing the current default version

id: The ID of the (major) version, also acts as the release number links: A list containing one link that points to the current version of the API

status: Status of the version, one of CURRENT, SUPPORTED, DEPRECATED

min_version: The current, maximum supported (major.minor) version of API.

version: Minimum supported (major.minor) version of API.

Module contents

ironic.api.middleware package

Submodules

ironic.api.middleware.auth_public_routes module

class ironic.api.middleware.auth_public_routes.**AuthPublicRoutes**(*app*, *auth*, *public_api_routes=None*)

Bases: object

A wrapper on authentication middleware.

Does not perform verification of authentication tokens for public routes in the API.

ironic.api.middleware.json_ext module

class ironic.api.middleware.json_ext.**JsonExtensionMiddleware**(*app*)

Bases: object

Simplified processing of .json extension.

Previously Ironic API used the guess_content_type_from_ext feature. It was never needed, as we never allowed non-JSON content types anyway. Now that it is removed, this middleware strips .json extension for backward compatibility.

ironic.api.middleware.parsable_error module

Middleware to replace the plain text message body of an error response with one formatted so the client can parse it.

Based on pecan.middleware.errordocument

class ironic.api.middleware.parsable_error.**ParsableErrorMiddleware**(*app*)

Bases: object

Replace error body with something the client can parse.

Module contents

class ironic.api.middleware.**AuthPublicRoutes**(*app*, *auth*, *public_api_routes=None*)

Bases: object

A wrapper on authentication middleware.

Does not perform verification of authentication tokens for public routes in the API.

class ironic.api.middleware.**JsonExtensionMiddleware**(*app*)

Bases: object

Simplified processing of .json extension.

Previously Ironic API used the guess_content_type_from_ext feature. It was never needed, as we never allowed non-JSON content types anyway. Now that it is removed, this middleware strips .json extension for backward compatibility.

class ironic.api.middleware.**ParsableErrorMiddleware**(*app*)
Bases: object

Replace error body with something the client can parse.

Submodules

ironic.api.app module

class ironic.api.app.**IronicCORS**(*application*, **args*, ***kwargs*)
Bases: oslo_middleware.cors.CORS

Ironic-specific CORS class

Were adding the Ironic-specific version headers to the list of simple headers in order that a request bearing those headers might be accepted by the Ironic REST API.

simple_headers = ['Accept', 'Accept-Language', 'Content-Type', 'Cache-Control', 'Content-Language', 'Expires', 'Last-Modified', 'Pragma', 'X-Auth-Token', 'X-OpenStack-Ironic-API-Maximum-Version', 'X-OpenStack-Ironic-API-Minimum-Version', 'X-OpenStack-Ironic-API-Version']

class ironic.api.app.**VersionSelectorApplication**
Bases: object

ironic.api.app.**get_pecan_config**()

ironic.api.app.**setup_app**(*pecan_config=None*, *extra_hooks=None*)

ironic.api.config module

ironic.api.functions module

class ironic.api.functions.**FunctionArgument**(*name*, *datatype*, *mandatory*, *default*)
Bases: object

An argument definition of an api entry

datatype
Data type

default
Default value if argument is omitted

mandatory
True if the argument is mandatory

name
argument name

resolve_type(*registry*)

class ironic.api.functions.**FunctionDefinition**(*func*)

Bases: object

An api entry definition

arguments
The function arguments (list of *FunctionArgument*)

body_type
If the body carry the datas of a single argument, its type

doc
Function documentation

extra_options
Dictionnary of protocol-specific options.

static get(*func*)
Returns the *FunctionDefinition* of a method.

get_arg(*name*)
Returns a *FunctionArgument* from its name

ignore_extra_args
True if extra arguments should be ignored, NOT inserted in the kwargs of the function and not raise UnknownArgument exceptions

name
Function name

resolve_types(*registry*)

return_type
Return type

set_arg_types(*argspec*, *arg_types*)

set_options(*body=None*, *ignore_extra_args=False*, *status_code=200*, *rest_content_types=('json', 'xml')*, ***extra_options*)

status_code
Status code

ironic.api.functions.**getargspec**(*f*)

ironic.api.functions.**iswsmefunction**(*f*)

ironic.api.functions.**sig**
alias of *ironic.api.functions.signature*

class ironic.api.functions.**signature**(**types*, ***options*)

Bases: object

Decorator that specify the argument types of an exposed function.

Parameters

- **return_type** Type of the value returned by the function
- **argN** Type of the Nth argument
- **body** If the function takes a final argument that is supposed to be the request body by itself, its type.

- **status_code** HTTP return status code of the function.
- **ignore_extra_args** Allow extra/unknow arguments (default to False)

Most of the time this decorator is not supposed to be used directly, unless you are not using WSME on top of another framework.

If an adapter is used, it will provide either a specialised version of this decororator, either a new decorator named @wsexpose that takes the same parameters (it will in addition expose the function, hence its name).

ironic.api.functions.**wrapfunc**(*f*)

ironic.api.hooks module

class ironic.api.hooks.**ConfigHook**
Bases: pecan.hooks.PecanHook

Attach the config object to the request so controllers can get to it.

before(*state*)
Override this method to create a hook that gets called after routing, but before the request gets passed to your controller.

Parameters **state** The Pecan state object for the current request.

class ironic.api.hooks.**ContextHook**(*public_api_routes*)
Bases: pecan.hooks.PecanHook

Configures a request context and attaches it to the request.

after(*state*)
Override this method to create a hook that gets called after the request has been handled by the controller.

Parameters **state** The Pecan state object for the current request.

before(*state*)
Override this method to create a hook that gets called after routing, but before the request gets passed to your controller.

Parameters **state** The Pecan state object for the current request.

class ironic.api.hooks.**DBHook**
Bases: pecan.hooks.PecanHook

Attach the dbapi object to the request so controllers can get to it.

before(*state*)
Override this method to create a hook that gets called after routing, but before the request gets passed to your controller.

Parameters **state** The Pecan state object for the current request.

class ironic.api.hooks.**NoExceptionTracebackHook**
Bases: pecan.hooks.PecanHook

Workaround rpc.common: deserialize_remote_exception.

deserialize_remote_exception builds rpc exception traceback into error message which is then sent to the client. Such behavior is a security concern so this hook is aimed to cut-off traceback from the error message.

after(*state*)
Override this method to create a hook that gets called after the request has been handled by the controller.

Parameters **state** The Pecan `state` object for the current request.

class ironic.api.hooks.**PublicUrlHook**
Bases: pecan.hooks.PecanHook

Attach the right public_url to the request.

Attach the right public_url to the request so resources can create links even when the API service is behind a proxy or SSL terminator.

before(*state*)
Override this method to create a hook that gets called after routing, but before the request gets passed to your controller.

Parameters **state** The Pecan `state` object for the current request.

class ironic.api.hooks.**RPCHook**
Bases: pecan.hooks.PecanHook

Attach the rpcapi object to the request so controllers can get to it.

before(*state*)
Override this method to create a hook that gets called after routing, but before the request gets passed to your controller.

Parameters **state** The Pecan `state` object for the current request.

ironic.api.hooks.**policy_deprecation_check**()

ironic.api.method module

ironic.api.method.**body**(*body_arg*)
Decorator which places HTTP request body JSON into a method argument

Parameters **body_arg** Name of argument to populate with body JSON

ironic.api.method.**expose**(*status_code=None*)

ironic.api.method.**format_exception**(*excinfo*, *debug=False*)
Extract informations that can be sent to the client.

ironic.api.wsgi module

WSGI script for Ironic API, installed by pbr.

ironic.api.wsgi.**initialize_wsgi_app**(*argv=['/home/zuul/src/opendev.org/openstack/ironic/.tox/pdf-docs/bin/sphinx-build', '-W', '-b', 'latex', 'doc/source', 'doc/build/pdf']*)

Module contents

ironic.cmd package

Submodules

ironic.cmd.api module

The Ironic Service API.

ironic.cmd.api.**main**()

ironic.cmd.conductor module

The Ironic Management Service

ironic.cmd.conductor.**issue_startup_warnings**(*conf*)

ironic.cmd.conductor.**main**()

ironic.cmd.conductor.**warn_about_unsafe_shred_parameters**(*conf*)

ironic.cmd.dbsync module

Run storage database migration.

class ironic.cmd.dbsync.**DBCommand**

Bases: object

check_obj_versions(*ignore_missing_tables=False*)

Check the versions of objects.

Check that the object versions are compatible with this release of ironic. It does this by comparing the objects .version field in the database, with the expected versions of these objects.

Returns None if compatible; a string describing the issue otherwise.

create_schema()

online_data_migrations()

revision()

stamp()

upgrade()

version()

ironic.cmd.dbsync.**add_command_parsers**(*subparsers*)

ironic.cmd.dbsync.**main**()

ironic.cmd.singleprocess module

ironic.cmd.singleprocess.**main**()

ironic.cmd.status module

class ironic.cmd.status.**Checks**

Bases: oslo_upgradecheck.upgradecheck.UpgradeCommands

Upgrade checks for the ironic-status upgrade check command

Upgrade checks should be added as separate methods in this class and added to _upgrade_checks tuple.

ironic.cmd.status.**main**()

Module contents

ironic.common package

Subpackages

ironic.common.glance_service package

Submodules

ironic.common.glance_service.image_service module

class ironic.common.glance_service.image_service.**GlanceImageService**(*client=None*, *context=None*)

Bases: object

call(*method*, **args*, ***kwargs*)

Call a glance client method.

If we get a connection error, retry the request according to CONF.num_retries.

Parameters

- **method** The method requested to be called.
- **args** A list of positional arguments for the method called
- **kwargs** A dict of keyword arguments for the method called

Raises GlanceConnectionFailed

download(*image_href*, *data=None*)
Calls out to Glance for data and writes data.

Parameters

- **image_href** The opaque image identifier.
- **data** (Optional) File object to write data to.

show(*image_href*)
Returns a dict with image data for the given opaque image id.

Parameters **image_href** The opaque image identifier.

Returns A dict containing image metadata.

Raises ImageNotFound

Raises ImageUnacceptable if the image status is not active

swift_temp_url(*image_info*)
Generate a no-auth Swift temporary URL.

This function will generate (or return the cached one if temp URL cache is enabled) the temporary Swift URL using the image id from Glance and the config options: swift_endpoint_url, swift_api_version, swift_account and swift_container. The temporary URL will be valid for swift_temp_url_duration seconds. This allows Ironic to download a Glance image without passing around an auth_token.

Parameters **image_info** The return from a GET request to Glance for a certain image_id. Should be a dictionary, with keys like name and checksum. See https://docs.openstack.org/glance/latest/user/glanceapi.html for examples.

Returns A signed Swift URL from which an image can be downloaded, without authentication.

Raises InvalidParameterValue if Swift config options are not set correctly.

Raises MissingParameterValue if a required parameter is not set.

Raises ImageUnacceptable if the image info from Glance does not have an image ID.

class ironic.common.glance_service.image_service.**TempUrlCacheElement**(*url*, *url_expires_at*)

Bases: tuple

url
Alias for field number 0

url_expires_at
Alias for field number 1

ironic.common.glance_service.image_service.**check_image_service**(*func*)
Creates a glance client if doesnt exists and calls the function.

ironic.common.glance_service.service_utils module

ironic.common.glance_service.service_utils.**is_glance_image**(*image_href*)

ironic.common.glance_service.service_utils.**is_image_active**(*image*)
Check the image status.

This check is needed in case the Glance image is stuck in queued status or pending_delete.

ironic.common.glance_service.service_utils.**is_image_available**(*context*, *image*)
Check image availability.

This check is needed in case Nova and Glance are deployed without authentication turned on.

ironic.common.glance_service.service_utils.**parse_image_id**(*image_href*)
Parse an image id from image href.

Parameters **image_href** href of an image

Returns image id parsed from image_href

Raises *InvalidImageRef* when input image href is invalid

ironic.common.glance_service.service_utils.**translate_from_glance**(*image*)

Module contents

Submodules

ironic.common.args module

ironic.common.args.**and_valid**(**validators*)
Validates that every supplied validator passes

The value returned from each validator is passed as the value to the next one.

Parameters

- **name** Name of the argument
- **value** A value

Returns The value transformed through every supplied validator

Raises The error from the first failed validator

ironic.common.args.**boolean**(*name*, *value*)
Validate that the value is a string representing a boolean

Parameters

- **name** Name of the argument
- **value** A string value

Returns The boolean representation of the value, or None if value is None

Raises InvalidParameterValue if the value cannot be converted to a boolean

ironic.common.args.**dict_valid**(***validators*)
Return a validator function which validates dict fields

Validators will replace the value with the validation result. Any dict item which has no validator is ignored. When a key is missing in the value then the corresponding validator will not be run.

Param validators dict where the key is a dict key to validate and the value is a validator function to run on that value

Returns validator function which takes name and value arguments

ironic.common.args.**integer**(*name*, *value*)
Validate that the value represents an integer

Parameters

- **name** Name of the argument
- **value** A value representing an integer

Returns The value as an int, or None if value is None

Raises InvalidParameterValue if the value does not represent an integer

ironic.common.args.**mac_address**(*name*, *value*)
Validate that the value represents a MAC address

Parameters

- **name** Name of the argument
- **value** A string value representing a MAC address

Returns The value as a normalized MAC address, or None if value is None

Raises InvalidParameterValue if the value is not a valid MAC address

ironic.common.args.**name**(*name*, *value*)
Validate that the value is a logical name

Parameters

- **name** Name of the argument
- **value** A logical name string value

Returns The value, or None if value is None

Raises InvalidParameterValue if the value is not a valid logical name

ironic.common.args.**or_valid**(**validators*)
Validates if at least one supplied validator passes

Parameters

- **name** Name of the argument
- **value** A value

Returns The value returned from the first successful validator

Raises The error from the last validator when every validation fails

ironic.common.args.**patch**(*name*, *value*, *, *schema={'items': {'additionalProperties': False, 'properties': {'op': {'enum': ['add', 'replace', 'remove'], 'type': 'string'}, 'path': {'pattern': '^(/[\\w-]+)+$', 'type': 'string'}, 'value': {}}, 'required': ['op', 'path'], 'type': 'object'}, 'type': 'array'}*)

Validate a patch API operation

ironic.common.args.**schema**(*schema*)

Return a validator function which validates the value with jsonschema

Param schema dict representing jsonschema to validate with

Returns validator function which takes name and value arguments

ironic.common.args.**string**(*name*, *value*)

Validate that the value is a string

Parameters

- **name** Name of the argument
- **value** A string value

Returns The string value, or None if value is None

Raises InvalidParameterValue if the value is not a string

ironic.common.args.**string_list**(*name*, *value*)

Validate and convert comma delimited string to a list.

Parameters

- **name** Name of the argument
- **value** A comma separated string of values

Returns A list of unique values (lower-cased), maintaining the same order, or None if value is None

Raises InvalidParameterValue if the value is not a string

ironic.common.args.**types**(**types*)

Return a validator function which checks the value is one of the types

Param types one or more types to use for the isinstance test

Returns validator function which takes name and value arguments

ironic.common.args.**uuid**(*name*, *value*)

Validate that the value is a UUID

Parameters

- **name** Name of the argument
- **value** A UUID string value

Returns The value, or None if value is None

Raises InvalidParameterValue if the value is not a valid UUID

ironic.common.args.**uuid_or_name**(*name*, *value*)

Validate that the value is a UUID or logical name

Parameters

- **name** Name of the argument
- **value** A UUID or logical name string value

Returns The value, or None if value is None

Raises InvalidParameterValue if the value is not a valid UUID or logical name

ironic.common.args.**validate**(**args*, ***kwargs*)
Decorator which validates and transforms function arguments

ironic.common.boot_devices module

Mapping of boot devices used when requesting the system to boot from an alternate device.

The options presented were based on the IPMItool chassis bootdev command. You can find the documentation at: http://linux.die.net/man/1/ipmitool

NOTE: This module does not include all the options from ipmitool because they dont make sense in the limited context of Ironic right now.

ironic.common.boot_devices.**BIOS = 'bios'**
Boot into BIOS setup

ironic.common.boot_devices.**CDROM = 'cdrom'**
Boot from CD/DVD

ironic.common.boot_devices.**DISK = 'disk'**
Boot from default Hard-drive

ironic.common.boot_devices.**FLOPPY = 'floppy'**
Boot from a floppy drive

ironic.common.boot_devices.**ISCSIBOOT = 'iscsiboot'**
Boot from iSCSI volume

ironic.common.boot_devices.**PXE = 'pxe'**
Boot from PXE boot

ironic.common.boot_devices.**SAFE = 'safe'**
Boot from default Hard-drive, request Safe Mode

ironic.common.boot_devices.**WANBOOT = 'wanboot'**
Boot from Wide Area Network

ironic.common.boot_modes module

Mapping of boot modes used when requesting the system to boot using alternative firmware interfaces.

The options presented were based on the Redfish protocol capabilities, specifically on the BootSourceOverrideMode property.

ironic.common.boot_modes.**LEGACY_BIOS = 'bios'**
Boot over legacy PC BIOS firmware interface

ironic.common.boot_modes.**UEFI = 'uefi'**
Boot over Unified Extensible Firmware Interface (UEFI) firmware interface

ironic.common.cinder module

ironic.common.cinder.**attach_volumes**(*task*, *volume_list*, *connector*)

Attach volumes to a node.

Enumerate through the provided list of volumes and attach the volumes to the node defined in the task utilizing the provided connector information.

If an attachment appears to already exist, we will skip attempting to attach the volume. If use of the volume fails, a user may need to remove any lingering pre-existing/unused attachment records since we have no way to validate if the connector profile data differs from what was provided to cinder.

Parameters

- **task** TaskManager instance representing the operation.
- **volume_list** List of volume_id UUID values representing volumes.
- **connector** Dictionary object representing the node sufficiently to attach a volume. This value can vary based upon the nodes configuration, capability, and ultimately the back-end storage driver. As cinder was designed around iSCSI, the ip and initiator keys are generally expected by cinder drivers. For FiberChannel, the key wwpns can be used with a list of port addresses. Some drivers support a multipath boolean key, although it is generally False. The host key is generally used for logging by drivers. Example:

```
{
'wwpns': ['list','of','port','wwns'],
'ip': 'ip address',
'initiator': 'initiator iqn',
'multipath': False,
'host': 'hostname',
}
```

Raises StorageError If storage subsystem exception is raised.

Returns

List of connected volumes, including volumes that were already connected to desired nodes. The returned list can be relatively consistent depending on the end storage driver that the volume is configured for, however the driver_volume_type key should not be relied upon as it is a free-form value returned by the driver. The accompanying data key contains the actual target details which will indicate either target WWNs and a LUN or a target portal and IQN. It also always contains volume ID in cinder and ironic. Except for these two IDs, each driver may return somewhat different data although the same keys are used if the target is FC or iSCSI, so any logic should be based upon the returned contents. For already attached volumes, the structure contains already_attached: True key-value pair. In such case, connection info for the node is already in the database, data structure contains only basic info of volume ID in cinder and ironic, so any logic based on that should retrieve it from the database. Example:

```
[{
'driver_volume_type': 'fibre_channel'
```

(continues on next page)

(continued from previous page)

```
'data': {
    'encrypted': False,
    'target_lun': 1,
    'target_wwn': ['1234567890123', '1234567890124'],
    'volume_id': '00000000-0000-0000-0000-000000000001',
    'ironic_volume_id':
    '11111111-0000-0000-0000-000000000001'}
},
{
'driver_volume_type': 'iscsi'
'data': {
    'target_iqn': 'iqn.2010-10.org.openstack:volume-000002',
    'target_portal': '127.0.0.0.1:3260',
    'volume_id': '00000000-0000-0000-0000-000000000002',
    'ironic_volume_id':
    '11111111-0000-0000-0000-000000000002',
    'target_lun': 2}
},
{
'already_attached': True
'data': {
    'volume_id': '00000000-0000-0000-0000-000000000002',
    'ironic_volume_id':
    '11111111-0000-0000-0000-000000000002'}
}]
```

ironic.common.cinder.**detach_volumes**(*task*, *volume_list*, *connector*, *allow_errors=False*)

Detach a list of volumes from a provided connector detail.

Enumerates through a provided list of volumes and issues detachment requests utilizing the connector information that describes the node.

Parameters

- **task** The TaskManager task representing the request.
- **volume_list** The list of volume id values to detach.
- **connector** Dictionary object representing the node sufficiently to attach a volume. This value can vary based upon the nodes configuration, capability, and ultimately the back-end storage driver. As cinder was designed around iSCSI, the ip and initiator keys are generally expected. For FiberChannel, the key wwpns can be used with a list of port addresses. Some drivers support a multipath boolean key, although it is generally False. The host key is generally used for logging by drivers. Example:

```
{
'wwpns': ['list','of','port','wwns']
'ip': 'ip address',
'initiator': 'initiator iqn',
'multipath': False,
'host': 'hostname'
```

(continues on next page)

(continued from previous page)

```
}
```

- **allow_errors** Boolean value governing if errors that are returned are treated as warnings instead of exceptions. Default False.

Raises StorageError

ironic.common.cinder.**get_client**(*context*, *auth_from_config=False*)
Get a cinder client connection.

Parameters

- **context** request context, instance of ironic.common.context.RequestContext
- **auth_from_config** (boolean) When True, use auth values from conf parameters

Returns A cinder client.

ironic.common.cinder.**is_volume_attached**(*node*, *volume*)
Check if a volume is attached to the supplied node.

Parameters

- **node** The object representing the node.
- **volume** The object representing the volume from cinder.

Returns Boolean indicating if the volume is attached. Returns True if cinder shows the volume as presently attached, otherwise returns False.

ironic.common.cinder.**is_volume_available**(*volume*)
Check if a volume is available for a connection.

Parameters **volume** The object representing the volume.

Returns Boolean if volume is available.

ironic.common.components module

Mapping of common hardware components of a computer system.

ironic.common.components.**CHASSIS = 'chassis'**
Chassis enclosing one or more hardware components

ironic.common.components.**DISK = 'disk'**
Storage drive

ironic.common.components.**NIC = 'nic'**
Network interface

ironic.common.components.**POWER = 'power'**
Power supply unit

ironic.common.components.**SYSTEM = 'system'**
Computing system

ironic.common.config module

ironic.common.config.**parse_args**(*argv*, *default_config_files=None*)

ironic.common.context module

class ironic.common.context.**RequestContext**(*is_public_api=False*, ***kwargs*)
Bases: oslo_context.context.RequestContext

Extends security contexts from the oslo.context library.

ensure_thread_contain_context()
Ensure threading contains context

For async/periodic tasks, the context of local thread is missing. Set it with request context and this is useful to log the request_id in log messages.

to_policy_values()
A dictionary of context attributes to enforce policy with.

oslo.policy enforcement requires a dictionary of attributes representing the current logged in user on which it applies policy enforcement. This dictionary defines a standard list of attributes that should be available for enforcement across services.

It is expected that services will often have to override this method with either deprecated values or additional attributes used by that service specific policy.

ironic.common.context.**get_admin_context**()
Create an administrator context.

ironic.common.dhcp_factory module

class ironic.common.dhcp_factory.**DHCPFactory**(***kwargs*)
Bases: object

clean_dhcp(*task*)
Clean up the DHCP BOOT options for this node.

Parameters **task** A TaskManager instance.

property provider

update_dhcp(*task*, *dhcp_opts*, *ports=None*)
Send or update the DHCP BOOT options for this node.

Parameters

- **task** A TaskManager instance.
- **dhcp_opts** this will be a list of dicts, e.g.

```
[{'opt_name': '67',
  'opt_value': 'pxelinux.0'},
 {'opt_name': '66',
  'opt_value': '123.123.123.456'}]
```

- **ports** A dict with keys ports and portgroups and dicts as values. Each dict has key/value pairs of the form <ironic UUID>:<neutron port UUID>. e.g.

```
{'ports': {'port.uuid': vif.id},
 'portgroups': {'portgroup.uuid': vif.id}}
```

  If the value is None, will get the list of ports/portgroups from the Ironic port/portgroup objects.

ironic.common.driver_factory module

class ironic.common.driver_factory.**BaseDriverFactory**
Bases: object

Discover, load and manage the drivers available.

This is subclassed to load both main drivers and extra interfaces.

get_driver(*name*)

items()
Iterator over pairs (name, instance).

property names
The list of driver names available.

class ironic.common.driver_factory.**HardwareTypesFactory**
Bases: *ironic.common.driver_factory.BaseDriverFactory*

class ironic.common.driver_factory.**InterfaceFactory**
Bases: *ironic.common.driver_factory.BaseDriverFactory*

class ironic.common.driver_factory.**NetworkInterfaceFactory**
Bases: *ironic.common.driver_factory.InterfaceFactory*

class ironic.common.driver_factory.**StorageInterfaceFactory**
Bases: *ironic.common.driver_factory.InterfaceFactory*

ironic.common.driver_factory.**all_interfaces**()
Get all interfaces for all interface types.

Returns Dictionary mapping interface type to dictionary mapping interface name to interface object.

ironic.common.driver_factory.**build_driver_for_task**(*task*)
Builds a composable driver for a given task.

Starts with a *BareDriver* object, and attaches implementations of the various driver interfaces to it. They come from separate driver factories and are configurable via the database.

Parameters **task** The task containing the node to build a driver for.

Returns A driver object for the task.

Raises DriverNotFound if node.driver could not be found in the ironic.hardware.types namespaces.

Raises InterfaceNotFoundInEntrypoint if some node interfaces are set to invalid or unsupported values.

Raises IncompatibleInterface the requested implementation is not compatible with it with the hardware type.

ironic.common.driver_factory.**check_and_update_node_interfaces**(*node*, *hw_type=None*)

Ensure that node interfaces (e.g. for creation or updating) are valid.

Updates (but doesnt save to the database) hardware interfaces with calculated defaults, if they are not provided.

This function is run on node updating and creation, as well as each time a driver instance is built for a node.

Parameters

- **node** node object to check and potentially update
- **hw_type** hardware type instance object; will be detected from node.driver if missing

Returns True if any changes were made to the node, otherwise False

Raises InterfaceNotFoundInEntrypoint on validation failure

Raises NoValidDefaultForInterface if the default value cannot be calculated and is not provided in the configuration

Raises DriverNotFound if the nodes hardware type is not found

ironic.common.driver_factory.**default_interface**(*hw_type*, *interface_type*, *driver_name=None*, *node=None*)

Calculate and return the default interface implementation.

Finds the first implementation that is supported by the hardware type and is enabled in the configuration.

Parameters

- **hw_type** hardware type instance object.
- **interface_type** type of the interface (e.g. boot).
- **driver_name** entrypoint name of the hw_type object. Is used for exception message.
- **node** the identifier of a node. If specified, is used for exception message.

Returns an entrypoint name of the calculated default implementation.

Raises InterfaceNotFoundInEntrypoint if the entry point was not found.

Raises NoValidDefaultForInterface if no default interface can be found.

ironic.common.driver_factory.**enabled_supported_interfaces**(*hardware_type*)

Get usable interfaces for a given hardware type.

For a given hardware type, find the intersection of enabled and supported interfaces for each interface type. This is the set of interfaces that are usable for this hardware type.

Parameters **hardware_type** The hardware type object to search.

Returns a dict mapping interface types to a list of enabled and supported interface names.

ironic.common.driver_factory.**get_hardware_type**(*hardware_type*)
Get a hardware type instance by name.

Parameters **hardware_type** the name of the hardware type to find

Returns An instance of ironic.drivers.hardware_type.AbstractHardwareType

Raises DriverNotFound if requested hardware type cannot be found

ironic.common.driver_factory.**get_interface**(*hw_type*, *interface_type*, *interface_name*)
Get interface implementation instance.

For hardware types also validates compatibility.

Parameters

- **hw_type** a hardware type instance.
- **interface_type** name of the interface type (e.g. boot).
- **interface_name** name of the interface implementation from an appropriate entry point (ironic.hardware.interfaces.<interface type>).

Returns instance of the requested interface implementation.

Raises InterfaceNotFoundInEntrypoint if the entry point was not found.

Raises IncompatibleInterface if hw_type is a hardware type and the requested implementation is not compatible with it.

ironic.common.driver_factory.**hardware_types**()
Get all hardware types.

Returns Dictionary mapping hardware type name to hardware type object.

ironic.common.driver_factory.**interfaces**(*interface_type*)
Get all interfaces for a given interface type.

Parameters **interface_type** String, type of interface to fetch for.

Returns Dictionary mapping interface name to interface object.

ironic.common.exception module

Ironic specific exceptions list.

exception ironic.common.exception.**AgentAPIError**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**AgentCommandTimeout**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**AgentConnectionFailed**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**AgentInProgress**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**AllocationAlreadyExists**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**AllocationDuplicateName**(*message=None*, ***kwargs*)

Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**AllocationFailed**(*message=None*, ***kwargs*)

Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**AllocationNotFound**(*message=None*, ***kwargs*)

Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**BIOSSettingAlreadyExists**(*message=None*, ***kwargs*)

Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**BIOSSettingListNotFound**(*message=None*, ***kwargs*)

Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**BIOSSettingNotFound**(*message=None*, ***kwargs*)

Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**BadRequest**(*message=None*, ***kwargs*)

Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**CatalogNotFound**(*message=None*, ***kwargs*)

Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**ChassisAlreadyExists**(*message=None*, ***kwargs*)

Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**ChassisNotEmpty**(*message=None*, ***kwargs*)

Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**ChassisNotFound**(*message=None*, ***kwargs*)

Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**ClientSideError**(*msg=None*, *status_code=400*, *faultcode='Client'*)

Bases: RuntimeError

property faultstring

exception ironic.common.exception.**CommunicationError**(*message=None*, ***kwargs*)

Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**ConductorAlreadyRegistered**(*message=None*, ***kwargs*)

Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**ConductorHardwareInterfacesAlreadyRegistered**(*message=None*, ***kwargs*)

Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**ConductorNotFound**(*message=None*, ***kwargs*)

Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**ConfigInvalid**(*message=None*, ***kwargs*)

Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**ConfigNotFound**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**Conflict**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

code = 409

exception ironic.common.exception.**ConsoleError**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**ConsoleSubprocessFailed**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.ConsoleError*

exception ironic.common.exception.**DHCPLoadError**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**DatabaseVersionTooOld**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**DeployTemplateAlreadyExists**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**DeployTemplateDuplicateName**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**DeployTemplateNotFound**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**DirectoryNotWritable**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**DracOperationError**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.DriverOperationError*

exception ironic.common.exception.**DriverLoadError**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**DriverNotFound**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**DriverNotFoundInEntrypoint**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.DriverNotFound*

exception ironic.common.exception.**DriverOperationError**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**Duplicate**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**DuplicateName**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**ExclusiveLockRequired**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotAuthorized*

exception ironic.common.exception.**FailedToCleanDHCPOpts**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**FailedToGetIPAddressOnPort**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**FailedToGetSensorData**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**FailedToParseSensorData**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**FailedToUpdateDHCPOptOnPort**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**FailedToUpdateMacOnPort**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**FileSystemNotSupported**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**Forbidden**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**GlanceConnectionFailed**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**HTTPForbidden**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotAuthorized*

exception ironic.common.exception.**HTTPNotFound**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**HardwareInspectionFailure**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**IBMCConnectionError**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.IBMCError*

exception ironic.common.exception.**IBMCError**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.DriverOperationError*

exception ironic.common.exception.**IPMIFailure**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**IRMCOperationError**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.DriverOperationError*

exception ironic.common.exception.**IRMCSharedFileSystemNotMounted**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.DriverOperationError*

exception ironic.common.exception.**IloOperationError**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.DriverOperationError*

exception ironic.common.exception.**IloOperationNotSupported**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.DriverOperationError*

exception ironic.common.exception.**ImageConvertFailed**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**ImageCreationFailed**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**ImageDownloadFailed**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**ImageNotAuthorized**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotAuthorized*

exception ironic.common.exception.**ImageNotFound**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**ImageRefValidationFailed**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**ImageUnacceptable**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**ImageUploadFailed**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**IncompatibleInterface**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.InvalidParameterValue*

exception ironic.common.exception.**IncompleteLookup**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**IncorrectConfiguration**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**InputFileError**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**InstanceAssociated**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**InstanceDeployFailure**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**InstanceNotFound**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**InstanceRescueFailure**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**InstanceUnrescueFailure**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**InsufficentMemory**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**InsufficientDiskSpace**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**InterfaceNotFoundInEntrypoint**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.InvalidParameterValue*

exception ironic.common.exception.**Invalid**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

code = 400

exception ironic.common.exception.**InvalidConductorGroup**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**InvalidDatapathID**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**InvalidDeployTemplate**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**InvalidEndpoint**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**InvalidIPAddress**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**InvalidIPv4Address**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**InvalidIdentity**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**InvalidImageRef**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.InvalidParameterValue*

exception ironic.common.exception.**InvalidInput**(*fieldname*, *value*, *msg=''*)
Bases: *ironic.common.exception.ClientSideError*

property faultstring

exception ironic.common.exception.**InvalidKickstartFile**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**InvalidKickstartTemplate**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**InvalidMAC**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**InvalidName**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**InvalidParameterValue**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**InvalidState**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**InvalidStateRequested**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**InvalidSwitchID**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**InvalidUUID**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**InvalidUuidOrName**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**KeystoneFailure**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**KeystoneUnauthorized**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**MACAlreadyExists**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**MissingParameterValue**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.InvalidParameterValue*

exception ironic.common.exception.**NetworkError**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**NoConsolePid**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.ConsoleError*

exception ironic.common.exception.**NoDriversLoaded**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**NoFreeConductorWorker**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.TemporaryFailure*

code = 503

exception ironic.common.exception.**NoFreeIPMITerminalPorts**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.TemporaryFailure*

exception ironic.common.exception.**NoFreePhysicalPorts**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**NoValidDefaultForInterface**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.InvalidParameterValue*

exception ironic.common.exception.**NoValidHost**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**NodeAlreadyExists**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**NodeAssociated**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.InvalidState*

exception ironic.common.exception.**NodeCleaningFailure**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**NodeConsoleNotEnabled**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**NodeHistoryNotFound**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**NodeInMaintenance**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**NodeIsRetired**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**NodeLocked**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**NodeMaintenanceFailure**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**NodeNotFound**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**NodeNotLocked**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**NodeProtected**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.HTTPForbidden*

exception ironic.common.exception.**NodeTagNotFound**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**NodeTraitNotFound**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**NodeVerifyFailure**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**NotAcceptable**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

code = 406

exception ironic.common.exception.**NotAuthorized**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

code = 403

exception ironic.common.exception.**NotFound**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

code = 404

exception ironic.common.exception.**NotificationPayloadError**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**NotificationSchemaKeyError**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**NotificationSchemaObjectError**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**OperationNotPermitted**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotAuthorized*

exception ironic.common.exception.**PasswordFileFailedToCreate**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**PatchError**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**PathNotFound**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**PortAlreadyExists**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**PortDuplicateName**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**PortNotFound**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**PortgroupAlreadyExists**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**PortgroupDuplicateName**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**PortgroupMACAlreadyExists**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**PortgroupNotEmpty**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**PortgroupNotFound**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**PortgroupPhysnetInconsistent**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**PowerStateFailure**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.InvalidState*

exception ironic.common.exception.**RedfishConnectionError**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.RedfishError*

exception ironic.common.exception.**RedfishError**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.DriverOperationError*

exception ironic.common.exception.**SNMPFailure**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.DriverOperationError*

exception ironic.common.exception.**ServiceUnavailable**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**StorageError**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**SwiftObjectNotFoundError**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.SwiftOperationError*

exception ironic.common.exception.**SwiftOperationError**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**TemporaryFailure**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

code = 503

exception ironic.common.exception.**Unauthorized**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**UnknownArgument**(*argname*, *msg=''*)
Bases: *ironic.common.exception.ClientSideError*

property faultstring

exception ironic.common.exception.**UnknownAttribute**(*fieldname*, *attributes*, *msg=''*)
Bases: *ironic.common.exception.ClientSideError*

add_fieldname(*name*)
Add a fieldname to concatenate the full name.

Add a fieldname so that the whole hierarchy is displayed. Successive calls to this method will prepend `name` to the hierarchy of names.

property faultstring

exception ironic.common.exception.**UnsupportedDriverExtension**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**VendorPassthruException**(*message=None*, ***kwargs*)
Bases: ironic_lib.exception.IronicException

exception ironic.common.exception.**VifAlreadyAttached**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**VifInvalidForAttach**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**VifNotAttached**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Invalid*

exception ironic.common.exception.**VolumeConnectorAlreadyExists**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**VolumeConnectorNotFound**(*message=None*, ***kwargs*)
Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**VolumeConnectorTypeAndIdAlreadyExists**(*message=None*, ***kwargs*)

Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**VolumeTargetAlreadyExists**(*message=None*, ***kwargs*)

Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**VolumeTargetBootIndexAlreadyExists**(*message=None*, ***kwargs*)

Bases: *ironic.common.exception.Conflict*

exception ironic.common.exception.**VolumeTargetNotFound**(*message=None*, ***kwargs*)

Bases: *ironic.common.exception.NotFound*

exception ironic.common.exception.**XClarityError**(*message=None*, ***kwargs*)

Bases: ironic_lib.exception.IronicException

ironic.common.faults module

Fault definitions.

ironic.common.faults.**CLEAN_FAILURE = 'clean failure'**

Node is moved to maintenance due to failure of a cleaning operation.

ironic.common.faults.**POWER_FAILURE = 'power failure'**

Node is moved to maintenance due to power synchronization failure.

ironic.common.faults.**RESCUE_ABORT_FAILURE = 'rescue abort failure'**

Node is moved to maintenance due to failure of cleaning up during rescue abort.

ironic.common.fsm module

class ironic.common.fsm.**FSM**

Bases: automaton.machines.FiniteMachine

An ironic state-machine class with some ironic specific additions.

add_state(*state*, *on_enter=None*, *on_exit=None*, *target=None*, *terminal=None*, *stable=False*)

Adds a given state to the state machine.

Parameters

- **stable** Use this to specify that this state is a stable/passive state. A state must have been previously defined as stable before it can be used as a target
- **target** The target state for state to go to. Before a state can be used as a target it must have been previously added and specified as stable

Further arguments are interpreted as for parent method add_state.

add_transition(*start*, *end*, *event*, *replace=False*)

Adds an allowed transition from start -> end for the given event.

Parameters

- **start** starting state

- **end** ending state
- **event** event that causes start state to transition to end state
- **replace** replace existing event instead of raising a `Duplicate` exception when the transition already exists.

initialize(*start_state=None*, *target_state=None*)
Initialize the FSM.

Parameters

- **start_state** the FSM is initialized to start from this state
- **target_state** if specified, the FSM is initialized to this target state. Otherwise use the default target state

is_stable(*state*)
Is the state stable?

Parameters **state** the state of interest

Raises InvalidState if the state is invalid

Returns True if it is a stable state; False otherwise

process_event(*event*, *target_state=None*)
process the event.

Parameters

- **event** the event to be processed
- **target_state** if specified, the final target state for the event. Otherwise, use the default target state

property target_state

ironic.common.hash_ring module

class ironic.common.hash_ring.**HashRingManager**(*use_groups=True*, *cache=True*)
Bases: `object`

get_ring(*driver_name*, *conductor_group*)

classmethod reset()

property ring

ironic.common.i18n module

oslo.i18n integration module.

See https://docs.openstack.org/oslo.i18n/latest/user/

ironic.common.image_service module

class ironic.common.image_service.**BaseImageService**
Bases: object

Provides retrieval of disk images.

abstract **download**(*image_href*, *image_file*)
Downloads image to specified location.

Parameters

- **image_href** Image reference.
- **image_file** File object to write data to.

Raises exception.ImageRefValidationFailed.

Raises exception.ImageDownloadFailed.

abstract **show**(*image_href*)
Get dictionary of image properties.

Parameters **image_href** Image reference.

Raises exception.ImageRefValidationFailed.

Returns dictionary of image properties. It has three of them: size, updated_at and properties. updated_at attribute is a naive UTC datetime object.

abstract **validate_href**(*image_href*)
Validate image reference.

Parameters **image_href** Image reference.

Raises exception.ImageRefValidationFailed.

Returns Information needed to further operate with an image.

class ironic.common.image_service.**FileImageService**
Bases: *ironic.common.image_service.BaseImageService*

Provides retrieval of disk images available locally on the conductor.

download(*image_href*, *image_file*)
Downloads image to specified location.

Parameters

- **image_href** Image reference.
- **image_file** File object to write data to.

Raises exception.ImageRefValidationFailed if source image file doesnt exist.

Raises exception.ImageDownloadFailed if exceptions were raised while writing to file or creating hard link.

show(*image_href*)
Get dictionary of image properties.

Parameters **image_href** Image reference.

Raises exception.ImageRefValidationFailed if image file specified doesnt exist.

Returns dictionary of image properties. It has three of them: size, updated_at and properties. updated_at attribute is a naive UTC datetime object.

validate_href(*image_href*)
Validate local image reference.

Parameters **image_href** Image reference.

Raises exception.ImageRefValidationFailed if source image file doesnt exist.

Returns Path to image file if it exists.

class ironic.common.image_service.**HttpImageService**
Bases: *ironic.common.image_service.BaseImageService*

Provides retrieval of disk images using HTTP.

download(*image_href*, *image_file*)
Downloads image to specified location.

Parameters

- **image_href** Image reference.
- **image_file** File object to write data to.

Raises exception.ImageRefValidationFailed if GET request returned response code not equal to 200.

Raises exception.ImageDownloadFailed if: * IOError happened during file write; * GET request failed.

show(*image_href*)
Get dictionary of image properties.

Parameters **image_href** Image reference.

Raises exception.ImageRefValidationFailed if: * HEAD request failed; * HEAD request returned response code not equal to 200; * Content-Length header not found in response to HEAD request.

Returns dictionary of image properties. It has three of them: size, updated_at and properties. updated_at attribute is a naive UTC datetime object.

validate_href(*image_href*, *secret=False*)
Validate HTTP image reference.

Parameters

- **image_href** Image reference.
- **secret** Specify if image_href being validated should not be shown in exception message.

Raises exception.ImageRefValidationFailed if HEAD request failed or returned response code not equal to 200.

Returns Response to HEAD request.

ironic.common.image_service.**get_image_service**(*image_href*, *client=None*, *context=None*)
Get image service instance to download the image.

Parameters

- **image_href** String containing href to get image service for.
- **client** Glance client to be used for download, used only if image_href is Glance href.
- **context** request context, used only if image_href is Glance href.

Raises exception.ImageRefValidationFailed if no image service can handle specified href.

Returns Instance of an image service class that is able to download specified image.

ironic.common.images module

Handling of VM disk images.

ironic.common.images.**converted_size**(*path*, *estimate=False*)

Get size of converted raw image.

The size of image converted to raw format can be growing up to the virtual size of the image.

Parameters

- **path** path to the image file.
- **estimate** Whether to estimate the size by scaling the original size

Returns For *estimate=False*, return the size of the raw image file. For *estimate=True*, return the size of the original image scaled by the configuration value *raw_image_growth_factor*.

ironic.common.images.**create_boot_iso**(*context*, *output_filename*, *kernel_href*, *ramdisk_href*, *deploy_iso_href=None*, *esp_image_href=None*, *root_uuid=None*, *kernel_params=None*, *boot_mode=None*, *inject_files=None*)

Creates a bootable ISO image for a node.

Given the hrefs for kernel, ramdisk, root partitions UUID and kernel cmdline arguments, this method fetches the kernel and ramdisk, and builds a bootable ISO image that can be used to boot up the baremetal node.

Parameters

- **context** context
- **output_filename** the absolute path of the output ISO file
- **kernel_href** URL or glance uuid of the kernel to use
- **ramdisk_href** URL or glance uuid of the ramdisk to use
- **deploy_iso_href** URL or glance UUID of the deploy ISO image to extract EFI system partition image. If not specified, the *esp_image_href* option must be present if UEFI-bootable ISO is desired.
- **esp_image_href** URL or glance UUID of FAT12/16/32-formatted EFI system partition image containing the EFI boot loader (e.g. GRUB2) for each hardware architecture to boot. This image will be written onto the ISO image. If not specified, the *deploy_iso_href* option is only required for building UEFI-bootable ISO.

- **kernel_params** a string containing whitespace separated values kernel cmdline arguments of the form K=V or K (optional).
- **inject_files** Mapping of local source file paths to their location on the final ISO image.

Boot_mode the boot mode in which the deploy is to happen.

Raises ImageCreationFailed, if creating boot ISO failed.

ironic.common.images.**create_esp_image_for_uefi**(*output_file*, *kernel*, *ramdisk*, *deploy_iso=None*, *esp_image=None*, *kernel_params=None*, *inject_files=None*)

Creates an ESP image on the specified file.

Copies the provided kernel, ramdisk and EFI system partition image (ESP) to a directory, generates the grub configuration file using kernel parameters and then generates a bootable ISO image for UEFI.

Parameters

- **output_file** the path to the file where the iso image needs to be created.
- **kernel** the kernel to use.
- **ramdisk** the ramdisk to use.
- **deploy_iso** deploy ISO image to extract EFI system partition image from. If not specified, the *esp_image* option is required.
- **esp_image** FAT12/16/32-formatted EFI system partition image containing the EFI boot loader (e.g. GRUB2) for each hardware architecture to boot. This image will be embedded into the ISO image. If not specified, the *deploy_iso* option is required.
- **kernel_params** a list of strings(each element being a string like K=V or K or combination of them like K1=V1,K2,) to be added as the kernel cmdline.
- **inject_files** Mapping of local source file paths to their location on the final ISO image.

Raises ImageCreationFailed, if image creation failed while copying files or while running command to generate iso.

ironic.common.images.**create_isolinux_image_for_bios**(*output_file*, *kernel*, *ramdisk*, *kernel_params=None*, *inject_files=None*)

Creates an isolinux image on the specified file.

Copies the provided kernel, ramdisk to a directory, generates the isolinux configuration file using the kernel parameters provided, and then generates a bootable ISO image.

Parameters

- **output_file** the path to the file where the iso image needs to be created.
- **kernel** the kernel to use.
- **ramdisk** the ramdisk to use.

- **kernel_params** a list of strings(each element being a string like K=V or K or combination of them like K1=V1,K2,) to be added as the kernel cmdline.
- **inject_files** Mapping of local source file paths to their location on the final ISO image.

Raises ImageCreationFailed, if image creation failed while copying files or while running command to generate iso.

ironic.common.images.**create_vfat_image**(*output_file*, *files_info=None*, *parameters=None*, *parameters_file='parameters.txt'*, *fs_size_kib=100*)

Creates the fat fs image on the desired file.

This method copies the given files to a root directory (optional), writes the parameters specified to the parameters file within the root directory (optional), and then creates a vfat image of the root directory.

Parameters

- **output_file** The path to the file where the fat fs image needs to be created.
- **files_info** A dict containing absolute path of file to be copied -> relative path within the vfat image. For example:

```
{
'/absolute/path/to/file' -> 'relative/path/within/root'
...
}
```

- **parameters** A dict containing key-value pairs of parameters.
- **parameters_file** The filename for the parameters file.
- **fs_size_kib** size of the vfat filesystem in KiB.

Raises ImageCreationFailed, if image creation failed while doing any of filesystem manipulation activities like creating dirs, mounting, creating filesystem, copying files, etc.

ironic.common.images.**download_size**(*context*, *image_href*, *image_service=None*)

ironic.common.images.**fetch**(*context*, *image_href*, *path*, *force_raw=False*)

ironic.common.images.**fetch_into**(*context*, *image_href*, *image_file*)

ironic.common.images.**force_raw_will_convert**(*image_href*, *path_tmp*)

ironic.common.images.**get_image_properties**(*context*, *image_href*, *properties='all'*)

Returns the values of several properties of an image

Parameters

- **context** context
- **image_href** href of the image
- **properties** the properties whose values are required. This argument is optional, default value is all, so if not specified all properties will be returned.

Returns a dict of the values of the properties. A property not on the glance metadata will have a value of None.

ironic.common.images.**get_source_format**(*image_href*, *path*)

ironic.common.images.**get_temp_url_for_glance_image**(*context*, *image_uuid*)
Returns the tmp url for a glance image.

Parameters

- **context** context
- **image_uuid** the UUID of the image in glance

Returns the tmp url for the glance image.

ironic.common.images.**image_show**(*context*, *image_href*, *image_service=None*)

ironic.common.images.**image_to_raw**(*image_href*, *path*, *path_tmp*)

ironic.common.images.**is_whole_disk_image**(*ctx*, *instance_info*)
Find out if the image is a partition image or a whole disk image.

Parameters

- **ctx** an admin context
- **instance_info** a nodes instance info dict

Returns True for whole disk images and False for partition images and None on no image_source or Error.

ironic.common.indicator_states module

Mapping of the indicator LED states.

ironic.common.indicator_states.**BLINKING = 'blinking'**
LED is blinking

ironic.common.indicator_states.**OFF = 'off'**
LED is off

ironic.common.indicator_states.**ON = 'on'**
LED is on

ironic.common.indicator_states.**UNKNOWN = 'unknown'**
LED state is not known

ironic.common.keystone module

Central place for handling Keystone authorization and service lookup.

ironic.common.keystone.**get_adapter**(*group*, ***adapter_kwargs*)
Loads adapter from options in a configuration file section.

The adapter_kwargs will be passed directly to keystoneauth1 Adapter and will override the values loaded from config. Consult keystoneauth1 docs for available adapter options.

Parameters **group** name of the config section to load adapter options from

ironic.common.keystone.**get_auth**(*group*, ***auth_kwargs*)

Loads auth plugin from options in a configuration file section.

The auth_kwargs will be passed directly to keystoneauth1 auth plugin and will override the values loaded from config. Note that the accepted kwargs will depend on auth plugin type as defined by [group]auth_type option. Consult keystoneauth1 docs for available auth plugins and their options.

Parameters **group** name of the config section to load auth plugin options from

ironic.common.keystone.**get_endpoint**(*group*, ***adapter_kwargs*)

Get an endpoint from an adapter.

The adapter_kwargs will be passed directly to keystoneauth1 Adapter and will override the values loaded from config. Consult keystoneauth1 docs for available adapter options.

Parameters **group** name of the config section to load adapter options from

Raises CatalogNotFound if the endpoint is not found

ironic.common.keystone.**get_service_auth**(*context*, *endpoint*, *service_auth*, *only_service_auth=False*)

Create auth plugin wrapping both user and service auth.

When properly configured and using auth_token middleware, requests with valid service auth will not fail if the user token is expired.

Ideally we would use the plugin provided by auth_token middleware however this plugin isnt serialized yet.

Parameters

- **context** The RequestContext instance from which the user auth_token is extracted.
- **endpoint** The requested endpoint to be utilized.
- **service_auth** The service authenticaiton credentals to be used.
- **only_service_auth** Boolean, default False. When set to True, the resulting Service token pair is generated as if it originates from the user itself. Useful to cast admin level operations which are launched by Ironic itself, as opposed to user initiated requests.

Returns Returns a service token via the ServiceTokenAuthWrapper class.

ironic.common.keystone.**get_session**(*group*, ***session_kwargs*)

Loads session object from options in a configuration file section.

The session_kwargs will be passed directly to keystoneauth1 Session and will override the values loaded from config. Consult keystoneauth1 docs for available options.

Parameters **group** name of the config section to load session options from

ironic.common.keystone.**ks_exceptions**(*f*)

Wraps keystoneclient functions and centralizes exception handling.

ironic.common.kickstart_utils module

ironic.common.kickstart_utils.**decode_and_extract_config_drive_iso**(*config_drive_iso_gz*)

ironic.common.kickstart_utils.**prepare_config_drive**(*task*, *config_drive_path='/var/lib/cloud/seed/config_drive'*)

Prepare config_drive for writing to kickstart file

ironic.common.kickstart_utils.**read_iso9600_config_drive**(*config_drive*)

Read config drive and store its contents in a dict

Parameters **config_drive** Config drive in iso9600 format

Returns A dict containing path as key and contents of the configdrive file as value.

ironic.common.molds module

ironic.common.molds.**get_configuration**(*task*, *url*)

Gets configuration mold from indicated location.

Parameters

- **task** A TaskManager instance.
- **url** URL of the configuration item to get.

Returns JSON configuration mold

Raises

- **IronicException** If using Swift storage and no authentication token found in tasks context.
- **HTTPError** If failed to complete HTTP request.

ironic.common.molds.**save_configuration**(*task*, *url*, *data*)

Store configuration mold to indicated location.

Parameters

- **task** A TaskManager instance.
- **name** URL of the configuration item to save to.
- **data** Content of JSON data to save.

Raises

- **IronicException** If using Swift storage and no authentication token found in tasks context.
- **HTTPError** If failed to complete HTTP request.

ironic.common.network module

ironic.common.network.**get_node_vif_ids**(*task*)
Get all VIF ids for a node.

This function does not handle multi node operations.

Parameters **task** a TaskManager instance.

Returns

A dict of Nodes neutron ports where keys are ports & portgroups and the values are dict of UUIDs and their associated VIFs, e.g.

```
{'ports': {'port.uuid': vif.id},
 'portgroups': {'portgroup.uuid': vif.id}}
```

ironic.common.network.**get_physnets_by_portgroup_id**(*task*, *portgroup_id*, *exclude_port=None*)
Return the set of physical networks associated with a portgroup.

Parameters

- **task** a TaskManager instance.
- **portgroup_id** ID of the portgroup.
- **exclude_port** A Port object to exclude from the determination of the portgroups physical network, or None.

Returns The set of physical networks associated with the portgroup. The set will contain zero or one physical networks.

Raises PortgroupPhysnetInconsistent if the portgroups ports are not assigned the same physical network.

ironic.common.network.**get_physnets_for_node**(*task*)
Return the set of physical networks for a node.

Returns the set of physical networks associated with a nodes ports. The physical network None is excluded from the set.

Parameters **task** a TaskManager instance

Returns A set of physical networks.

ironic.common.network.**get_portgroup_by_id**(*task*, *portgroup_id*)
Lookup a portgroup by ID on a task object.

Parameters

- **task** a TaskManager instance
- **portgroup_id** ID of the portgroup.

Returns A Portgroup object or None.

ironic.common.network.**get_ports_by_portgroup_id**(*task*, *portgroup_id*)
Lookup ports by their portgroup ID on a task object.

Parameters

- **task** a TaskManager instance
- **portgroup_id** ID of the portgroup.

Returns A list of Port objects.

ironic.common.network.**remove_vifs_from_node**(*task*)
Remove all vif attachment records from a node.

Parameters **task** a TaskManager instance.

ironic.common.neutron module

class ironic.common.neutron.**NeutronNetworkInterfaceMixin**
Bases: object

get_cleaning_network_uuid(*task*)

get_inspection_network_uuid(*task*)

get_provisioning_network_uuid(*task*)

get_rescuing_network_uuid(*task*)

validate_inspection(*task*)
Validate that the node has required properties for inspection.

Parameters **task** A TaskManager instance with the node being checked

Raises MissingParameterValue if node is missing one or more required parameters

Raises UnsupportedDriverExtension

ironic.common.neutron.**PHYSNET_PARAM_NAME = 'provider:physical_network'**
Name of the neutron network API physical network parameter.

ironic.common.neutron.**add_ports_to_network**(*task*, *network_uuid*, *security_groups=None*)
Create neutron ports to boot the ramdisk.

Create neutron ports for each pxe_enabled port on task.node to boot the ramdisk.

If the config option neutron.add_all_ports is set, neutron ports for non-pxe-enabled ports are also created these neutron ports will not have any assigned IP addresses.

Parameters

- **task** a TaskManager instance.
- **network_uuid** UUID of a neutron network where ports will be created.
- **security_groups** List of Security Groups UUIDs to be used for network.

Raises NetworkError

Returns a dictionary in the form {port.uuid: neutron_port[id]}

ironic.common.neutron.**get_client**(*token=None*, *context=None*, *auth_from_config=False*)
Retrieve a neutron client connection.

Parameters

- **context** request context, instance of ironic.common.context.RequestContext

- **auth_from_config** (boolean) When True, use auth values from conf parameters

Returns A neutron client.

ironic.common.neutron.**get_local_group_information**(*task*, *portgroup*)

Extract the portgroup information.

The information is returned in the form of:

```
{
    'id': portgroup.uuid,
    'name': portgroup.name,
    'bond_mode': portgroup.mode,
    'bond_properties': {
        'bond_propertyA': 'valueA',
        'bond_propertyB': 'valueB',
    }
}
```

Parameters

- **task** a task containing the Node object.
- **portgroup** Ironic portgroup object to extract data for.

Returns port group information as a dict

ironic.common.neutron.**get_neutron_port_data**(*port_id*, *vif_id*, *client=None*, *context=None*)

Gather Neutron port and network configuration

Query Neutron for port and network configuration, return whatever is available.

Parameters

- **port_id** ironic port/portgroup ID.
- **vif_id** Neutron port ID.
- **client** Optional a Neutron client object.
- **context** (ironic.common.context.RequestContext) request context

Raises NetworkError

Returns a dict holding network configuration information associated with this ironic or Neutron port.

ironic.common.neutron.**get_node_portmap**(*task*)

Extract the switch port information for the node.

The information is returned in the form of:

```
{
    port.uuid: {
        'switch_id': 'abc',
        'port_id': 'Po0/1',
        'other_llc_key': 'val'
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Parameters **task** a task containing the Node object.

Returns port information as a dict

ironic.common.neutron.**get_physnets_by_port_uuid**(*client*, *port_uuid*)
Return the set of physical networks associated with a neutron port.

Query the network to which the port is attached and return the set of physical networks associated with the segments in that network.

Parameters

- **client** A Neutron client object.
- **port_uuid** UUID of a Neutron port to query.

Returns A set of physical networks.

Raises NetworkError if the network query fails.

Raises InvalidParameterValue for missing network.

ironic.common.neutron.**is_smartnic_port**(*port_data*)
Check that the port is Smart NIC port

Parameters **port_data** an instance of ironic.objects.port.Port or port data as dict.

Returns A boolean to indicate port as Smart NIC port.

ironic.common.neutron.**remove_neutron_ports**(*task*, *params*)
Deletes the neutron ports matched by params.

Parameters

- **task** a TaskManager instance.
- **params** Dict of params to filter ports.

Raises NetworkError

ironic.common.neutron.**remove_ports_from_network**(*task*, *network_uuid*)
Deletes the neutron ports created for booting the ramdisk.

Parameters

- **task** a TaskManager instance.
- **network_uuid** UUID of a neutron network ports will be deleted from.

Raises NetworkError

ironic.common.neutron.**rollback_ports**(*task*, *network_uuid*)
Attempts to delete any ports created by cleaning/provisioning

Purposefully will not raise any exceptions so error handling can continue.

Parameters

- **task** a TaskManager instance.

- **network_uuid** UUID of a neutron network.

ironic.common.neutron.**unbind_neutron_port**(*port_id*, *client=None*, *context=None*, *reset_mac=True*)

Unbind a neutron port

Remove a neutron ports binding profile and host ID so that it returns to an unbound state.

Parameters

- **port_id** Neutron port ID.
- **client** Optional a Neutron client object.
- **context** (ironic.common.context.RequestContext) request context
- **reset_mac** reset mac address

Raises NetworkError

ironic.common.neutron.**update_neutron_port**(*context*, *port_id*, *attrs*, *client=None*)

Undate a neutron port

Uses neutron client from conf client to update a neutron client an unbound state.

Parameters

- **context** request context, instance of ironic.common.context.RequestContext
- **port_id** Neutron port ID.
- **attrs** The attributes to update on the port
- **client** Optional Neutron client

ironic.common.neutron.**update_port_address**(*port_id*, *address*, *context=None*)

Update a ports mac address.

Parameters

- **port_id** Neutron port id.
- **address** new MAC address.
- **context** (ironic.common.context.RequestContext) request context

Raises FailedToUpdateMacOnPort

ironic.common.neutron.**validate_network**(*uuid_or_name*, *net_type='network'*, *context=None*)

Check that the given network is present.

Parameters

- **uuid_or_name** network UUID or name
- **net_type** human-readable network type for error messages
- **context** (ironic.common.context.RequestContext) request context

Returns network UUID

Raises MissingParameterValue if uuid_or_name is empty

Raises NetworkError on failure to contact Neutron

Raises InvalidParameterValue for missing or duplicated network

ironic.common.neutron.**validate_port_info**(*node*, *port*)
Check that port contains enough information for deploy.

Neutron network interface requires that local_link_information field is filled before we can use this port.

Parameters

- **node** Ironic node object.
- **port** Ironic port object.

Returns True if port info is valid, False otherwise.

ironic.common.neutron.**wait_for_host_agent**(*client*, *host_id*, *target_state='up'*)
Wait for neutron agent to become target state

Parameters

- **client** A Neutron client object.
- **host_id** Agent host_id
- **target_state** up: wait for up status, down: wait for down status

Returns boolean indicates the agent state matches param value target_state_up.

Raises exception.Invalid if target_state is not valid.

Raises exception.NetworkError if host status didnt match the required status after max retry attempts.

ironic.common.neutron.**wait_for_port_status**(*client*, *port_id*, *status*)
Wait for port status to be the desired status

Parameters

- **client** A Neutron client object.
- **port_id** Neutron port_id
- **status** Ports target status, can be ACTIVE, DOWN etc.

Returns boolean indicates that the port status matches the required value passed by param status.

Raises InvalidParameterValue if the port does not exist.

Raises exception.NetworkError if port status didnt match the required status after max retry attempts.

ironic.common.nova module

ironic.common.nova.**power_update**(*context*, *server_uuid*, *target_power_state*)
Creates and sends power state change for the provided server_uuid.

Parameters

- **context** request context, instance of ironic.common.context.RequestContext
- **server_uuid** The uuid of the node whose power state changed.

- **target_power_state** Targeted power state change i.e POWER_ON or POWER_OFF

Returns A boolean which indicates if the power update was executed successfully (mainly for testing purposes).

ironic.common.policy module

Policy Engine For Ironic.

ironic.common.policy.**authorize**(*rule*, *target*, *creds*, **args*, ***kwargs*)
A shortcut for policy.Enforcer.authorize()

Checks authorization of a rule against the target and credentials, and raises an exception if the rule is not defined. Always returns true if CONF.auth_strategy is not keystone.

ironic.common.policy.**check**(*rule*, *target*, *creds*, **args*, ***kwargs*)
A shortcut for policy.Enforcer.enforce()

Checks authorization of a rule against the target and credentials and returns True or False.

ironic.common.policy.**check_policy**(*rule*, *target*, *creds*, **args*, ***kwargs*)
Configuration aware role policy check wrapper.

Checks authorization of a rule against the target and credentials and returns True or False. Always returns true if CONF.auth_strategy is not keystone.

ironic.common.policy.**get_enforcer**()
Provides access to the single instance of Policy enforcer.

ironic.common.policy.**get_oslo_policy_enforcer**()

ironic.common.policy.**init_enforcer**(*policy_file=None*, *rules=None*, *default_rule=None*, *use_conf=True*)
Synchronously initializes the policy enforcer

Parameters

- **policy_file** Custom policy file to use, if none is specified, *CONF.oslo_policy.policy_file* will be used.
- **rules** Default dictionary / Rules to use. It will be considered just in the first instantiation.
- **default_rule** Default rule to use, CONF.oslo_policy.policy_default_rule will be used if none is specified.
- **use_conf** Whether to load rules from config file.

ironic.common.policy.**list_policies**()

ironic.common.profiler module

ironic.common.profiler.**setup**(*name*, *host='0.0.0.0'*)
Setup OSprofiler notifier and enable profiling.

Parameters

- **name** name of the service that will be profiled
- **host** hostname or host IP address that the service will be running on. By default host will be set to 0.0.0.0, but specifying host name / address usage is highly recommended.

Raises **TypeError** in case of invalid connection string for a notifier backend, which is set in osprofiler.initializer.init_from_conf.

ironic.common.profiler.**trace_cls**(*name*, ***kwargs*)
Wrap the OSProfiler trace_cls decorator

Wrap the OSProfiler trace_cls decorator so that it will not try to patch the class unless OSProfiler is present and enabled in the config

Parameters

- **name** The name of action. For example, wsgi, rpc, db, etc..
- **kwargs** Any other keyword args used by profiler.trace_cls

ironic.common.pxe_utils module

class ironic.common.pxe_utils.**TFTPImageCache**
Bases: *ironic.drivers.modules.image_cache.ImageCache*

ironic.common.pxe_utils.**build_deploy_pxe_options**(*task*, *pxe_info*, *mode='deploy'*, *ipxe_enabled=False*)

ironic.common.pxe_utils.**build_extra_pxe_options**(*task*, *ramdisk_params=None*)

ironic.common.pxe_utils.**build_instance_pxe_options**(*task*, *pxe_info*, *ipxe_enabled=False*)

ironic.common.pxe_utils.**build_kickstart_config_options**(*task*)
Build the kickstart template options for a node

This method builds the kickstart template options for a node, given all the required parameters.

The options should then be passed to pxe_utils.create_kickstart_config to create the actual config files.

Parameters **task** A TaskManager object

Returns A dictionary of kickstart options to be used in the kickstart template.

ironic.common.pxe_utils.**build_pxe_config_options**(*task*, *pxe_info*, *service=False*, *ipxe_enabled=False*, *ramdisk_params=None*)
Build the PXE config options for a node

This method builds the PXE boot options for a node, given all the required parameters.

The options should then be passed to pxe_utils.create_pxe_config to create the actual config files.

Parameters

- **task** A TaskManager object
- **pxe_info** a dict of values to set on the configuration file
- **service** if True, build service mode pxe config for netboot-ed user image and skip adding deployment image kernel and ramdisk info to PXE options.
- **ipxe_enabled** Default false boolean to indicate if ipxe is in use by the caller.
- **ramdisk_params** the parameters to be passed to the ramdisk. as kernel command-line arguments.

Returns A dictionary of pxe options to be used in the pxe bootfile template.

ironic.common.pxe_utils.**build_service_pxe_config**(*task*, *instance_image_info*, *root_uuid_or_disk_id*, *ramdisk_boot=False*, *ipxe_enabled=False*, *is_whole_disk_image=None*, *anaconda_boot=False*)

ironic.common.pxe_utils.**cache_ramdisk_kernel**(*task*, *pxe_info*, *ipxe_enabled=False*)
Fetch the necessary kernels and ramdisks for the instance.

ironic.common.pxe_utils.**clean_up_pxe_config**(*task*, *ipxe_enabled=False*)
Clean up the TFTP environment for the tasks node.

Parameters **task** A TaskManager instance.

ironic.common.pxe_utils.**clean_up_pxe_env**(*task*, *images_info*, *ipxe_enabled=False*)
Cleanup PXE environment of all the images in images_info.

Cleans up the PXE environment for the mentioned images in images_info.

Parameters

- **task** a TaskManager object
- **images_info** A dictionary of images whose keys are the image names to be cleaned up (kernel, ramdisk, etc) and values are a tuple of identifier and absolute path.

ironic.common.pxe_utils.**create_ipxe_boot_script**()
Render the iPXE boot script into the HTTP root directory

ironic.common.pxe_utils.**create_pxe_config**(*task*, *pxe_options*, *template=None*, *ipxe_enabled=False*)
Generate PXE configuration file and MAC address links for it.

This method will generate the PXE configuration file for the tasks node under a directory named with the UUID of that node. For each MAC address or DHCP IP address (port) of that node, a symlink for the configuration file will be created under the PXE configuration directory, so regardless of which port boots first theyll get the same PXE configuration. If grub2 bootloader is in use, then its configuration will be created based on DHCP IP address in the form nn.nn.nn.nn.

Parameters

- **task** A TaskManager instance.

- **pxe_options** A dictionary with the PXE configuration parameters.
- **template** The PXE configuration template. If no template is given the node specific template will be used.

ironic.common.pxe_utils.**dhcp_options_for_instance**(*task*, *ipxe_enabled=False*, *url_boot=False*, *ip_version=None*)

Retrieves the DHCP PXE boot options.

Parameters

- **task** A TaskManager instance.
- **ipxe_enabled** Default false boolean that signals if iPXE formatting should be returned by the method for DHCP server configuration.
- **url_boot** Default false boolean to inform the method if a URL should be returned to boot the node. If [pxe]ip_version is set to *6*, then this option has no effect as url_boot form is required by DHCPv6 standards.
- **ip_version** The IP version of options to return as values differ by IP version. Default to [pxe]ip_version. Possible options are integers 4 or 6.

Returns Dictionary to be sent to the networking service describing the DHCP options to be set.

ironic.common.pxe_utils.**ensure_tree**(*path*)

ironic.common.pxe_utils.**get_file_path_from_label**(*node_uuid*, *root_dir*, *label*)

Generate absolute paths to various images from their name(label)

This method generates absolute file system path on the conductor where various images need to be placed. For example the kickstart template, file and stage2 squashfs.img needs to be placed in the ipxe_root_dir since they will be transferred by anaconda ramdisk over http(s). The generated paths will be added to the image_info dictionary as values.

Parameters

- **node_uuid** the UUID of the node
- **root_dir** Directory in which the image must be placed
- **label** Name of the image

ironic.common.pxe_utils.**get_http_url_path_from_label**(*http_url*, *node_uuid*, *label*)

Generate http url path to various image artifacts

This method generates http(s) urls for various image artifacts int the webserver root. The generated urls will be added to the pxe_options dict and used to render pxe/ipxe configuration templates.

Parameters

- **http_url** URL to access the root of the webserver
- **node_uuid** the UUID of the node
- **label** Name of the image

ironic.common.pxe_utils.**get_image_info**(*node*, *mode='deploy'*, *ipxe_enabled=False*)

Generate the paths for TFTP files for deploy or rescue images.

This method generates the paths for the deploy (or rescue) kernel and deploy (or rescue) ramdisk.

Parameters

- **node** a node object
- **mode** Label indicating a deploy or rescue operation being carried out on the node. Supported values are deploy and rescue. Defaults to deploy, indicating deploy operation is being carried out.
- **ipxe_enabled** A default False boolean value to tell the method if the caller is using iPXE.

Returns a dictionary whose keys are the names of the images (deploy_kernel, deploy_ramdisk, or rescue_kernel, rescue_ramdisk) and values are the absolute paths of them.

Raises MissingParameterValue, if deploy_kernel/deploy_ramdisk or rescue_kernel/rescue_ramdisk is missing in nodes driver_info.

ironic.common.pxe_utils.**get_instance_image_info**(*task*, *ipxe_enabled=False*)

Generate the paths for TFTP files for instance related images.

This method generates the paths for instance kernel and instance ramdisk. This method also updates the node, so caller should already have a non-shared lock on the node.

Parameters

- **task** A TaskManager instance containing node and context.
- **ipxe_enabled** Default false boolean to indicate if ipxe is in use by the caller.

Returns a dictionary whose keys are the names of the images (kernel, ramdisk) and values are the absolute paths of them. If its a whole disk image or node is configured for localboot, it returns an empty dictionary.

ironic.common.pxe_utils.**get_kernel_ramdisk_info**(*node_uuid*, *driver_info*, *mode='deploy'*, *ipxe_enabled=False*)

Get href and tftp path for deploy or rescue kernel and ramdisk.

Parameters

- **node_uuid** UUID of the node
- **driver_info** Nodes driver_info dict
- **mode** A label to indicate whether paths for deploy or rescue ramdisk are being requested. Supported values are deploy rescue. Defaults to deploy, indicating deploy paths will be returned.
- **ipxe_enabled** A default False boolean value to tell the method if the caller is using iPXE.

Returns a dictionary whose keys are deploy_kernel and deploy_ramdisk or rescue_kernel and rescue_ramdisk and whose values are the absolute paths to them.

Note: driver_info should be validated outside of this method.

ironic.common.pxe_utils.**get_pxe_config_file_path**(*node_uuid*, *ipxe_enabled=False*)

Generate the path for the nodes PXE configuration file.

Parameters

- **node_uuid** the UUID of the node.

- **ipxe_enabled** A default False boolean value to tell the method if the caller is using iPXE.

Returns The path to the nodes PXE configuration file.

ironic.common.pxe_utils.**get_volume_pxe_options**(*task*)
Identify volume information for iPXE template generation.

ironic.common.pxe_utils.**is_ipxe_enabled**(*task*)
Return true if ipxe is set.

Parameters **task** A TaskManager object

Returns boolean true if the task driver instance is the iPXE driver.

ironic.common.pxe_utils.**parse_driver_info**(*node*, *mode='deploy'*)
Gets the driver specific Node deployment info.

This method validates whether the driver_info property of the supplied node contains the required information for this driver to deploy images to, or rescue, the node.

Parameters

- **node** a single Node.
- **mode** Label indicating a deploy or rescue operation being carried out on the node. Supported values are deploy and rescue. Defaults to deploy, indicating deploy operation is being carried out.

Returns A dict with the driver_info values.

Raises MissingParameterValue

ironic.common.pxe_utils.**place_common_config**()
Place template generated config which is not node specific.

Currently places the initial grub config for grub network boot.

ironic.common.pxe_utils.**place_loaders_for_boot**(*base_path*)
Place configured bootloaders from the host OS.

Example: grubaa64.efi:/path/to/grub-aarch64.efi,

Parameters **base_path** Destination path where files should be copied to.

ironic.common.pxe_utils.**prepare_instance_kickstart_config**(*task*, *image_info*, *anaconda_boot=False*)
Prepare to boot anaconda ramdisk by generating kickstart file

Parameters

- **task** a task from TaskManager.
- **image_info** a dict of values of instance image metadata to set on the configuration file.
- **anaconda_boot** if the boot is to a anaconda ramdisk configuration.

ironic.common.pxe_utils.**prepare_instance_pxe_config**(*task*, *image_info*, *iscsi_boot=False*, *ramdisk_boot=False*, *ipxe_enabled=False*, *anaconda_boot=False*)

Prepares the config file for PXE boot

Parameters

- **task** a task from TaskManager.
- **image_info** a dict of values of instance image metadata to set on the configuration file.
- **iscsi_boot** if boot is from an iSCSI volume or not.
- **ramdisk_boot** if the boot is to a ramdisk configuration.
- **ipxe_enabled** Default false boolean to indicate if ipxe is in use by the caller.
- **anaconda_boot** if the boot is to a anaconda ramdisk configuration.

Returns None

ironic.common.pxe_utils.**validate_boot_parameters_for_trusted_boot**(*node*)

Check if boot parameters are valid for trusted boot.

ironic.common.pxe_utils.**validate_kickstart_file**(*ks_cfg*)

Check if the kickstart file is valid

Parameters **ks_cfg** Contents of kickstart file to validate

Raises InvalidKickstartFile

ironic.common.pxe_utils.**validate_kickstart_template**(*ks_template*)

Validate the kickstart template

Parameters **ks_template** Path to the kickstart template

Raises InvalidKickstartTemplate

ironic.common.raid module

ironic.common.raid.**SAS = 'sas'**

Serial Attached SCSI

ironic.common.raid.**SATA = 'sata'**

Serial AT Attachment

ironic.common.raid.**SCSI = 'scsi'**

Small Computer System Interface

ironic.common.raid.**filter_target_raid_config**(*node*, *create_root_volume=True*, *create_nonroot_volumes=True*)

Filter the target raid config based on root volume creation

This method can be used by any raid interface which wants to filter out target raid config based on condition whether the root volume will be created or not.

Parameters

- **node** a node object
- **create_root_volume** A boolean default value True governing if the root volume is returned else root volumes will be filtered out.
- **create_nonroot_volumes** A boolean default value True governing if the non root volume is returned else non-root volumes will be filtered out.

Raises MissingParameterValue, if node.target_raid_config is missing or was found to be empty after skipping root volume and/or non-root volumes.

Returns It will return filtered target_raid_config

ironic.common.raid.**get_logical_disk_properties**(*raid_config_schema*)

Get logical disk properties from RAID configuration schema.

This method reads the logical properties and their textual description from the schema that is passed.

Parameters **raid_config_schema** A dictionary which is the schema to be used for getting properties that may be specified for the logical disk.

Returns A dictionary containing the logical disk properties as keys and a textual description for them as values.

ironic.common.raid.**update_raid_info**(*node*, *raid_config*)

Update the nodes information based on the RAID config.

This method updates the nodes information to make use of the configured RAID for scheduling purposes (through properties[capabilities] and properties[local_gb]) and deploying purposes (using properties[root_device]).

Parameters

- **node** a node object
- **raid_config** The dictionary containing the current RAID configuration.

Raises InvalidParameterValue, if raid_config has more than one root volume or if node.properties[capabilities] is malformed.

ironic.common.raid.**validate_configuration**(*raid_config*, *raid_config_schema*)

Validates the RAID configuration passed using JSON schema.

This method validates a RAID configuration against a RAID configuration schema.

Parameters

- **raid_config** A dictionary containing RAID configuration information
- **raid_config_schema** A dictionary which is the schema to be used for validation.

Raises InvalidParameterValue, if validation of the RAID configuration fails.

ironic.common.release_mappings module

ironic.common.release_mappings.**get_object_versions**(*releases=None*, *objects=None*)
Gets the supported versions for all objects.

Supported versions are from the RELEASE_MAPPINGs.

Parameters

- **releases** a list of release names; if empty/None, versions from all releases are returned (the default).
- **objects** a list of names of objects of interest. If empty/None, versions of all objects are returned (the default).

Returns a dictionary where the key is the object name and the value is a set of supported versions.

ironic.common.rpc module

class ironic.common.rpc.**RequestContextSerializer**(*base*)
Bases: oslo_messaging.serializer.Serializer

deserialize_context(*context*)
Deserialize a dictionary into a request context.

Parameters **ctxt** Request context dictionary

Returns Deserialized form of entity

deserialize_entity(*context*, *entity*)
Deserialize something from primitive form.

Parameters

- **ctxt** Request context, in deserialized form
- **entity** Primitive to be deserialized

Returns Deserialized form of entity

serialize_context(*context*)
Serialize a request context into a dictionary.

Parameters **ctxt** Request context

Returns Serialized form of context

serialize_entity(*context*, *entity*)
Serialize something to primitive form.

Parameters

- **ctxt** Request context, in deserialized form
- **entity** Entity to be serialized

Returns Serialized form of entity

ironic.common.rpc.**cleanup**()

ironic.common.rpc.**get_allowed_exmods**()

ironic.common.rpc.**get_client**(*target*, *version_cap=None*, *serializer=None*)

ironic.common.rpc.**get_sensors_notifier**(*service=None*, *host=None*, *publisher_id=None*)

ironic.common.rpc.**get_server**(*target*, *endpoints*, *serializer=None*)

ironic.common.rpc.**get_transport_url**(*url_str=None*)

ironic.common.rpc.**get_versioned_notifier**(*publisher_id=None*)

ironic.common.rpc.**init**(*conf*)

ironic.common.rpc.**set_defaults**(*control_exchange*)

ironic.common.rpc.**set_global_manager**(*manager*)

ironic.common.rpc_service module

class ironic.common.rpc_service.**RPCService**(*host*, *manager_module*, *manager_class*)

Bases: oslo_service.service.Service

handle_signal()

Add a signal handler for SIGUSR1.

The handler ensures that the manager is not deregistered when it is shutdown.

start()

Start a service.

stop()

Stop a service.

Parameters **graceful** indicates whether to wait for all threads to finish or terminate them instantly

wait_for_start()

ironic.common.service module

ironic.common.service.**ensure_rpc_transport**(*conf=<oslo_config.cfg.ConfigOpts object>*)

ironic.common.service.**prepare_command**(*argv=None*)

Prepare any Ironic command for execution.

Sets up configuration and logging, registers objects.

ironic.common.service.**prepare_service**(*name*, *argv=None*, *conf=<oslo_config.cfg.ConfigOpts object>*)

Prepare an Ironic service executable.

In addition to what *prepare_command* does, set up guru meditation reporting and profiling.

ironic.common.service.**process_launcher**()

ironic.common.states module

Mapping of bare metal node states.

Setting the node *power_state* is handled by the conductors power synchronization thread. Based on the power state retrieved from the driver for the node, the state is set to POWER_ON or POWER_OFF, accordingly. Should this fail, the *power_state* value is left unchanged, and the node is placed into maintenance mode.

The *power_state* can also be set manually via the API. A failure to change the state leaves the current state unchanged. The node is NOT placed into maintenance mode in this case.

ironic.common.states.**ACTIVE = 'active'**
Node is successfully deployed and associated with an instance.

ironic.common.states.**ADOPTFAIL = 'adopt failed'**
Node failed to complete the adoption process.

This state is the resulting state of a node that failed to complete adoption, potentially due to invalid or incompatible information being defined for the node.

ironic.common.states.**ADOPTING = 'adopting'**
Node is being adopted.

This provision state is intended for use to move a node from MANAGEABLE to ACTIVE state to permit designation of nodes as being managed by Ironic, however deployed previously by external means.

ironic.common.states.**AVAILABLE = 'available'**
Node is available for use and scheduling.

This state is replacing the NOSTATE state used prior to Kilo.

ironic.common.states.**CLEANFAIL = 'clean failed'**
Node failed cleaning. This requires operator intervention to resolve.

ironic.common.states.**CLEANING = 'cleaning'**
Node is being automatically cleaned to prepare it for provisioning.

ironic.common.states.**CLEANWAIT = 'clean wait'**
Node is waiting for a clean step to be finished.

This will be the nodes *provision_state* while the node is waiting for the driver to finish a cleaning step.

ironic.common.states.**DELETED = 'deleted'**
Node tear down was successful.

In Juno, target_provision_state was set to this value during node tear down.

In Kilo, this will be a transitory value of provision_state, and never represented in target_provision_state.

ironic.common.states.**DELETE_ALLOWED_STATES = ('manageable', 'enroll', 'adopt failed')**
States in which node deletion is allowed.

ironic.common.states.**DELETING = 'deleting'**
Node is actively being torn down.

ironic.common.states.**DEPLOY = 'deploy'**
Node is successfully deployed and associated with an instance. This is an alias for ACTIVE.

ironic.common.states.**DEPLOYDONE = 'deploy complete'**
Node was successfully deployed.

This is mainly a target provision state used during deployment. A successfully deployed node should go to ACTIVE status.

ironic.common.states.**DEPLOYFAIL = 'deploy failed'**
Node deployment failed.

ironic.common.states.**DEPLOYING = 'deploying'**
Node is ready to receive a deploy request, or is currently being deployed.

A node will have its *provision_state* set to DEPLOYING briefly before it receives its initial deploy request. It will also move to this state from DEPLOYWAIT after the callback is triggered and deployment is continued (disk partitioning and image copying).

ironic.common.states.**DEPLOYWAIT = 'wait call-back'**
Node is waiting to be deployed.

This will be the node *provision_state* while the node is waiting for the driver to finish deployment.

ironic.common.states.**ENROLL = 'enroll'**
Node is enrolled.

This state indicates that Ironic is aware of a node, but is not managing it.

ironic.common.states.**ERROR = 'error'**
An error occurred during node processing.

The *last_error* attribute of the node details should contain an error message.

ironic.common.states.**FASTTRACK_LOOKUP_ALLOWED_STATES = frozenset({'available', 'clean wait', 'cleaning', 'deploying', 'enroll', 'inspect wait', 'inspecting', 'manageable', 'rescue wait', 'rescuing', 'wait call-back'})**
States where API lookups are permitted with fast track enabled.

ironic.common.states.**INSPECTFAIL = 'inspect failed'**
Node inspection failed.

ironic.common.states.**INSPECTING = 'inspecting'**
Node is under inspection.

This is the provision state used when inspection is started. A successfully inspected node shall transition to MANAGEABLE state. For asynchronous inspection, node shall transition to INSPECTWAIT state.

ironic.common.states.**INSPECTWAIT = 'inspect wait'**
Node is under inspection.

This is the provision state used when an asynchronous inspection is in progress. A successfully inspected node shall transition to MANAGEABLE state.

ironic.common.states.**LOOKUP_ALLOWED_STATES = frozenset({'clean wait', 'cleaning', 'deploying', 'inspect wait', 'inspecting', 'rescue wait', 'rescuing', 'wait call-back'})**
States when API lookups are normally allowed for nodes.

ironic.common.states.**MANAGEABLE = 'manageable'**
Node is in a manageable state.

This state indicates that Ironic has verified, at least once, that it had sufficient information to manage the hardware. While in this state, the node is not available for provisioning (it must be in the AVAILABLE state for that).

ironic.common.states.**NOSTATE = None**
No state information.

This state is used with power_state to represent a lack of knowledge of power state, and in target_*_state fields when there is no target.

ironic.common.states.**POWER_OFF = 'power off'**
Node is powered off.

ironic.common.states.**POWER_ON = 'power on'**
Node is powered on.

ironic.common.states.**REBOOT = 'rebooting'**
Node is rebooting.

ironic.common.states.**REBUILD = 'rebuild'**
Node is to be rebuilt.

This is not used as a state, but rather as a verb when changing the nodes provision_state via the REST API.

ironic.common.states.**RESCUE = 'rescue'**
Node is in rescue mode.

ironic.common.states.**RESCUEFAIL = 'rescue failed'**
Node rescue failed.

ironic.common.states.**RESCUEWAIT = 'rescue wait'**
Node is waiting on an external callback.

This will be the node *provision_state* while the node is waiting for the driver to finish rescuing the node.

ironic.common.states.**RESCUING = 'rescuing'**
Node is in process of being rescued.

ironic.common.states.**SOFT_POWER_OFF = 'soft power off'**
Node is in the process of soft power off.

ironic.common.states.**SOFT_REBOOT = 'soft rebooting'**
Node is rebooting gracefully.

ironic.common.states.**STABLE_STATES = ('enroll', 'manageable', 'available', 'active', 'error', 'rescue')**
States that will not transition unless receiving a request.

ironic.common.states.**STUCK_STATES_TREATED_AS_FAIL = ('deploying', 'cleaning', 'verifying', 'inspecting', 'adopting', 'rescuing', 'unrescuing', 'deleting')**
States that cannot be resumed once a conductor dies.

If a node gets stuck with one of these states for some reason (eg. conductor goes down when executing task), node will be moved to fail state.

ironic.common.states.**UNDEPLOY = 'undeploy'**
Node tear down process has started. This is an alias for DELETED.

ironic.common.states.**UNRESCUEFAIL = 'unrescue failed'**
Node unrescue failed.

ironic.common.states.**UNRESCUING = 'unrescuing'**
Node is being restored from rescue mode (to active state).

ironic.common.states.**UNSTABLE_STATES = ('deploying', 'wait call-back', 'cleaning', 'clean wait', 'verifying', 'deleting', 'inspecting', 'inspect wait', 'adopting', 'rescuing', 'rescue wait', 'unrescuing')**
States that can be changed without external request.

ironic.common.states.**UPDATE_ALLOWED_STATES = ('deploy failed', 'inspecting', 'inspect failed', 'inspect wait', 'clean failed', 'error', 'verifying', 'adopt failed', 'rescue failed', 'unrescue failed')**
Transitional states in which we allow updating a node.

ironic.common.states.**VERBS = {'abort': 'abort', 'active': 'deploy', 'adopt': 'adopt', 'clean': 'clean', 'deleted': 'delete', 'deploy': 'deploy', 'inspect': 'inspect', 'manage': 'manage', 'provide': 'provide', 'rescue': 'rescue', 'undeploy': 'delete', 'unrescue': 'unrescue'}**
Mapping of state-changing events that are PUT to the REST API

This is a mapping of target states which are PUT to the API, eg, PUT /v1/node/states/provision {target: active}

The dict format is: {target string used by the API: internal verb}

This provides a reference set of supported actions, and in the future may be used to support renaming these actions.

ironic.common.states.**VERIFYING = 'verifying'**
Node power management credentials are being verified.

ironic.common.states.**on_enter**(*new_state*, *event*)
Used to log when entering a state.

ironic.common.states.**on_exit**(*old_state*, *event*)
Used to log when a state is exited.

ironic.common.swift module

class ironic.common.swift.**SwiftAPI**
Bases: object

API for communicating with Swift.

connection = None
Underlying Swift connection object.

create_object(*container*, *obj*, *filename*, *object_headers=None*)
Uploads a given file to Swift.

Parameters

- **container** The name of the container for the object.

- **obj** The name of the object in Swift
- **filename** The file to upload, as the object data
- **object_headers** the headers for the object to pass to Swift

Returns The Swift UUID of the object

Raises SwiftOperationError, if any operation with Swift fails.

delete_object(*container*, *obj*)
Deletes the given Swift object.

Parameters

- **container** The name of the container in which Swift object is placed.
- **obj** The name of the object in Swift to be deleted.

Raises SwiftObjectNotFoundError, if object is not found in Swift.

Raises SwiftOperationError, if operation with Swift fails.

get_temp_url(*container*, *obj*, *timeout*)
Returns the temp url for the given Swift object.

Parameters

- **container** The name of the container in which Swift object is placed.
- **obj** The name of the Swift object.
- **timeout** The timeout in seconds after which the generated url should expire.

Returns The temp url for the object.

Raises SwiftOperationError, if any operation with Swift fails.

head_object(*container*, *obj*)
Retrieves the information about the given Swift object.

Parameters

- **container** The name of the container in which Swift object is placed.
- **obj** The name of the object in Swift

Returns The information about the object as returned by Swift clients head_object call.

Raises SwiftOperationError, if operation with Swift fails.

update_object_meta(*container*, *obj*, *object_headers*)
Update the metadata of a given Swift object.

Parameters

- **container** The name of the container in which Swift object is placed.
- **obj** The name of the object in Swift
- **object_headers** the headers for the object to pass to Swift

Raises SwiftOperationError, if operation with Swift fails.

ironic.common.swift.**get_swift_session**()

ironic.common.utils module

Utilities and helper functions.

ironic.common.utils.**check_dir**(*directory_to_check=None*, *required_space=1*)
Check a directory is usable.

This function can be used by drivers to check that directories they need to write to are usable. This should be called from the drivers init function. This function checks that the directory exists and then calls check_dir_writable and check_dir_free_space. If directory_to_check is not provided the default is to use the temp directory.

Parameters

- **directory_to_check** the directory to check.
- **required_space** amount of space to check for in MiB.

Raises PathNotFound if directory can not be found

Raises DirectoryNotWritable if user is unable to write to the directory

Raises *InsufficientDiskSpace* if free space is < required space

ironic.common.utils.**create_link_without_raise**(*source*, *link*)

ironic.common.utils.**execute**(**cmd*, ***kwargs*)
Convenience wrapper around oslos execute() method.

Parameters

- **cmd** Passed to processutils.execute.
- **use_standard_locale** True | False. Defaults to False. If set to True, execute command with standard locale added to environment variables.

Returns (stdout, stderr) from process execution

Raises UnknownArgumentError

Raises ProcessExecutionError

ironic.common.utils.**fast_track_enabled**(*node*)

ironic.common.utils.**file_has_content**(*path*, *content*, *hash_algo='sha256'*)
Checks that content of the file is the same as provided reference.

Parameters

- **path** path to file
- **content** reference content to check against
- **hash_algo** hashing algo from hashlib to use, default is sha256

Returns True if the hash of reference content is the same as the hash of files content, False otherwise

ironic.common.utils.**file_mime_type**(*path*)
Gets a mime type of the given file.

ironic.common.utils.**get_updated_capabilities**(*current_capabilities*, *new_capabilities*)
Returns an updated capability string.

This method updates the original (or current) capabilities with the new capabilities. The original capabilities would typically be from a nodes properties[capabilities]. From new_capabilities, any new capabilities are added, and existing capabilities may have their values updated. This updated capabilities string is returned.

Parameters

- **current_capabilities** Current capability string
- **new_capabilities** the dictionary of capabilities to be updated.

Returns An updated capability string. with new_capabilities.

Raises ValueError, if current_capabilities is malformed or if new_capabilities is not a dictionary

ironic.common.utils.**is_fips_enabled**()

Check if FIPS mode is enabled in the system.

ironic.common.utils.**is_hostname_safe**(*hostname*)

Old check for valid logical node names.

Retained for compatibility with REST API < 1.10.

Nominally, checks that the supplied hostname conforms to:

- http://en.wikipedia.org/wiki/Hostname
- http://tools.ietf.org/html/rfc952
- http://tools.ietf.org/html/rfc1123

In practice, this check has several shortcomings and errors that are more thoroughly documented in bug #1468508.

Parameters **hostname** The hostname to be validated.

Returns True if valid. False if not.

ironic.common.utils.**is_memory_insufficent**(*raise_if_fail=False*)

Checks available system memory and holds the deployment process.

Evaluates the current system memory available, meaning can be allocated to a process by the kernel upon allocation request, and delays the execution until memory has been freed, or until it has timed out.

This method will issue a sleep, if the amount of available memory is insufficent. This is configured using the `[DEFAULT]minimum_memory_wait_time` and the `[DEFAULT]minimum_memory_wait_retries`.

Parameters **raise_if_fail** Default False, but if set to true an InsufficentMemory exception is raised upon insufficent memory.

Returns True if the check has timed out. Otherwise None is returned.

Raises InsufficentMemory if the raise_if_fail parameter is set to True.

ironic.common.utils.**is_regex_string_in_file**(*path*, *string*)

ironic.common.utils.**is_valid_datapath_id**(*datapath_id*)

Verify the format of an OpenFlow datapath_id.

Check if a datapath_id is valid and contains 16 hexadecimal digits. Datapath ID format: the lower 48-bits are for a MAC address, while the upper 16-bits are implementer-defined.

Parameters **datapath_id** OpenFlow datapath_id to be validated.

Returns True if valid. False if not.

ironic.common.utils.**is_valid_logical_name**(*hostname*)
Determine if a logical name is valid.

The logical name may only consist of RFC3986 unreserved characters, to wit:

ALPHA / DIGIT / - / . / _ / ~

ironic.common.utils.**is_valid_no_proxy**(*no_proxy*)
Check no_proxy validity

Check if no_proxy value that will be written to environment variable by ironic-python-agent is valid.

Parameters **no_proxy** the value that requires validity check. Expected to be a comma-separated list of host names, IP addresses and domain names (with optional :port).

Returns True if no_proxy is valid, False otherwise.

ironic.common.utils.**mount**(*src*, *dest*, **args*)
Mounts a device/image file on specified location.

Parameters

- **src** the path to the source file for mounting
- **dest** the path where it needs to be mounted.
- **args** a tuple containing the arguments to be passed to mount command.

Raises processutils.ProcessExecutionError if it failed to run the process.

ironic.common.utils.**parse_instance_info_capabilities**(*node*)
Parse the instance_info capabilities.

One way of having these capabilities set is via Nova, where the capabilities are defined in the Flavor extra_spec and passed to Ironic by the Nova Ironic driver.

NOTE: Although our API fully supports JSON fields, to maintain the backward compatibility with Juno the Nova Ironic driver is sending it as a string.

Parameters **node** a single Node.

Raises InvalidParameterValue if the capabilities string is not a dictionary or is malformed.

Returns A dictionary with the capabilities if found, otherwise an empty dictionary.

ironic.common.utils.**pop_node_nested_field**(*node*, *collection*, *field*, *default=None*)
Pop a value from a dictionary field of a node.

Parameters

- **node** Node object.
- **collection** Name of the field with the dictionary.

- **field** Nested field name.
- **default** The default value to return.

Returns The removed value or the default.

ironic.common.utils.**remove_large_keys**(*var*)
Remove specific keys from the var, recursing into dicts and lists.

ironic.common.utils.**render_template**(*template*, *params*, *is_file=True*, *strict=False*)
Renders Jinja2 template file with given parameters.

Parameters

- **template** full path to the Jinja2 template file
- **params** dictionary with parameters to use when rendering
- **is_file** whether template is file or string with template itself
- **strict** Enable strict template rendering. Default is False

Returns Rendered template

Raises jinja2.exceptions.UndefinedError

ironic.common.utils.**rmtree_without_raise**(*path*)

ironic.common.utils.**safe_rstrip**(*value*, *chars=None*)
Removes trailing characters from a string if that does not make it empty

Parameters

- **value** A string value that will be stripped.
- **chars** Characters to remove.

Returns Stripped value.

ironic.common.utils.**set_node_nested_field**(*node*, *collection*, *field*, *value*)
Set a value in a dictionary field of a node.

Parameters

- **node** Node object.
- **collection** Name of the field with the dictionary.
- **field** Nested field name.
- **value** New value.

ironic.common.utils.**tempdir**(***kwargs*)

ironic.common.utils.**umount**(*loc*, **args*)
Umounts a mounted location.

Parameters

- **loc** the path to be unmounted.
- **args** a tuple containing the arguments to be passed to the umount command.

Raises processutils.ProcessExecutionError if it failed to run the process.

ironic.common.utils.**unix_file_modification_datetime**(*file_name*)

ironic.common.utils.**validate_and_normalize_datapath_id**(*datapath_id*)
Validate an OpenFlow datapath_id and return normalized form.

Checks whether the supplied OpenFlow datapath_id is formally correct and normalize it to all lower case.

Parameters **datapath_id** OpenFlow datapath_id to be validated and normalized.

Returns Normalized and validated OpenFlow datapath_id.

Raises InvalidDatapathID If an OpenFlow datapath_id is not valid.

ironic.common.utils.**validate_and_normalize_mac**(*address*)
Validate a MAC address and return normalized form.

Checks whether the supplied MAC address is formally correct and normalize it to all lower case.

Parameters **address** MAC address to be validated and normalized.

Returns Normalized and validated MAC address.

Raises InvalidMAC If the MAC address is not valid.

ironic.common.utils.**validate_conductor_group**(*conductor_group*)

ironic.common.utils.**validate_network_port**(*port*, *port_name='Port'*)
Validates the given port.

Parameters

- **port** TCP/UDP port.
- **port_name** Name of the port.

Returns An integer port number.

Raises InvalidParameterValue, if the port is invalid.

ironic.common.utils.**wrap_ipv6**(*ip*)
Wrap the address in square brackets if its an IPv6 address.

ironic.common.utils.**write_to_file**(*path*, *contents*, *permission=None*)

ironic.common.wsgi_service module

class ironic.common.wsgi_service.**WSGIService**(*name*, *use_ssl=False*)
Bases: oslo_service.service.ServiceBase

Provides ability to launch ironic API from wsgi app.

reset()
Reset server greenpool size to default.

Returns None

start()
Start serving this service using loaded configuration.

Returns None

stop()
Stop serving this API.

Returns None

`wait()`
Wait for the service to stop serving this API.

Returns None

Module contents

ironic.conductor package

Submodules

ironic.conductor.allocations module

Functionality related to allocations.

`ironic.conductor.allocations.backfill_allocation(context, allocation, node_id)`
Assign the previously allocated node to the node allocation.

This is not the actual allocation process, but merely backfilling of allocation_uuid for a previously allocated node.

Parameters

- **context** an admin context
- **allocation** an allocation object associated with the node
- **node_id** An ID of the node.

Raises AllocationFailed if the node does not match the allocation

Raises NodeAssociated if the node is already associated with another instance or allocation.

Raises InstanceAssociated if the allocations UUID is already used on another node as instance_uuid.

Raises NodeNotFound if the node with the provided ID cannot be found.

`ironic.conductor.allocations.do_allocate(context, allocation)`
Process the allocation.

This call runs in a separate thread on a conductor. It finds suitable nodes for the allocation and reserves one of them.

This call does not raise exceptions since its designed to work asynchronously.

Parameters

- **context** an admin context
- **allocation** an allocation object

`ironic.conductor.allocations.verify_node_for_deallocation(node, allocation)`
Verify that allocation can be removed for the node.

Parameters

- **node** a node object
- **allocation** an allocation object associated with the node

ironic.conductor.base_manager module

Base conductor manager functionality.

class ironic.conductor.base_manager.**BaseConductorManager**(*host*, *topic*)

Bases: object

del_host(*deregister=True*)

init_host(*admin_context=None*)

Initialize the conductor host.

Parameters **admin_context** the admin context to pass to periodic tasks.

Raises RuntimeError when conductor is already running.

Raises NoDriversLoaded when no drivers are enabled on the conductor.

Raises DriverNotFound if a driver is enabled that does not exist.

Raises DriverLoadError if an enabled driver cannot be loaded.

Raises DriverNameConflict if a classic driver and a dynamic driver are both enabled and have the same name.

iter_nodes(*fields=None*, ***kwargs*)

Iterate over nodes mapped to this conductor.

Requests node set from and filters out nodes that are not mapped to this conductor.

Yields tuples (node_uuid, driver, conductor_group,) where is derived from fields argument, e.g.: fields=None means yielding (uuid, driver, conductor_group), fields=[foo] means yielding (uuid, driver, conductor_group, foo).

Parameters

- **fields** list of fields to fetch in addition to uuid, driver, and conductor_group
- **kwargs** additional arguments to pass to dbapi when looking for nodes

Returns generator yielding tuples of requested fields

prepare_host()

Prepares host for initialization

Prepares the conductor for basic operation by removing any existing transitory node power states and reservations which were previously held by this host.

Under normal operation, this is also when the initial database connectivity is established for the conductors normal operation.

ironic.conductor.cleaning module

Functionality related to cleaning.

ironic.conductor.cleaning.**continue_node_clean**(*task*)
Continue cleaning after finishing an async clean step.

This function calculates which step has to run next and passes control into do_next_clean_step.

Parameters **task** a TaskManager instance with an exclusive lock

ironic.conductor.cleaning.**do_next_clean_step**(*task*, *step_index*, *disable_ramdisk=None*)
Do cleaning, starting from the specified clean step.

Parameters

- **task** a TaskManager instance with an exclusive lock
- **step_index** The first clean step in the list to execute. This is the index (from 0) into the list of clean steps in the nodes driver_internal_info[clean_steps]. Is None if there are no steps to execute.
- **disable_ramdisk** Whether to skip booting ramdisk for cleaning.

ironic.conductor.cleaning.**do_node_clean**(*task*, *clean_steps=None*, *disable_ramdisk=False*)
Internal RPC method to perform cleaning of a node.

Parameters

- **task** a TaskManager instance with an exclusive lock on its node
- **clean_steps** For a manual clean, the list of clean steps to perform. Is None For automated cleaning (default). For more information, see the clean_steps parameter of `ConductorManager.do_node_clean()`.
- **disable_ramdisk** Whether to skip booting ramdisk for cleaning.

ironic.conductor.cleaning.**do_node_clean_abort**(*task*)
Internal method to abort an ongoing operation.

Parameters **task** a TaskManager instance with an exclusive lock

ironic.conductor.cleaning.**get_last_error**(*node*)

ironic.conductor.deployments module

Functionality related to deploying and undeploying.

ironic.conductor.deployments.**continue_node_deploy**(*task*)
Continue deployment after finishing an async deploy step.

This function calculates which step has to run next and passes control into do_next_deploy_step. On the first run, deploy steps and templates are also validated.

Parameters **task** a TaskManager instance with an exclusive lock

ironic.conductor.deployments.**do_next_deploy_step**(*task*, *step_index*)
Do deployment, starting from the specified deploy step.

Parameters

- **task** a TaskManager instance with an exclusive lock
- **step_index** The first deploy step in the list to execute. This is the index (from 0) into the list of deploy steps in the nodes driver_internal_info[deploy_steps]. Is None if there are no steps to execute.

ironic.conductor.deployments.**do_node_deploy**(*task*, *conductor_id=None*, *configdrive=None*, *deploy_steps=None*)

Prepare the environment and deploy a node.

ironic.conductor.deployments.**start_deploy**(*task*, *manager*, *configdrive=None*, *event='deploy'*, *deploy_steps=None*)

Start deployment or rebuilding on a node.

This function does not check the node suitability for deployment, its left up to the caller.

Parameters

- **task** a TaskManager instance.
- **manager** a ConductorManager to run tasks on.
- **configdrive** a configdrive, if requested.
- **event** event to process: deploy or rebuild.
- **deploy_steps** Optional deploy steps.

ironic.conductor.deployments.**validate_deploy_steps**(*task*)

Validate the deploy steps after the ramdisk learns about them.

ironic.conductor.deployments.**validate_node**(*task*, *event='deploy'*)

Validate that a node is suitable for deployment/rebuilding.

Parameters

- **task** a TaskManager instance.
- **event** event to process: deploy or rebuild.

Raises NodeInMaintenance, NodeProtected, InvalidStateRequested

ironic.conductor.manager module

Conduct all activity related to bare-metal deployments.

A single instance of *ironic.conductor.manager.ConductorManager* is created within the *ironic-conductor* process, and is responsible for performing all actions on bare metal resources (Chassis, Nodes, and Ports). Commands are received via RPCs. The conductor service also performs periodic tasks, eg. to monitor the status of active deployments.

Drivers are loaded via entrypoints by the *ironic.common.driver_factory* class. Each driver is instantiated only once, when the ConductorManager service starts. In this way, a single ConductorManager may use multiple drivers, and manage heterogeneous hardware.

When multiple *ConductorManager* are run on different hosts, they are all active and cooperatively manage all nodes in the deployment. Nodes are locked by each conductor when performing actions which change the state of that node; these locks are represented by the *ironic.conductor.task_manager.TaskManager* class.

A tooz.hashring.HashRing is used to distribute nodes across the set of active conductors which support each nodes driver. Rebalancing this ring can trigger various actions by each conductor, such as building or tearing down the TFTP environment for a node, notifying Neutron of a change, etc.

class ironic.conductor.manager.**ConductorManager**(*host*, *topic*)

Bases: *ironic.conductor.base_manager.BaseConductorManager*

Ironic Conductor manager main class.

RPC_API_VERSION = '1.55'

add_node_traits(***kwargs*)

change_node_boot_mode(***kwargs*)

change_node_power_state(***kwargs*)

change_node_secure_boot(***kwargs*)

continue_node_clean(*context*, *node_id*)

RPC method to continue cleaning a node.

This is useful for cleaning tasks that are async. When they complete, they call back via RPC, a new worker and lock are set up, and cleaning continues. This can also be used to resume cleaning on take_over.

Parameters

- **context** an admin context.
- **node_id** the id or uuid of a node.

Raises InvalidStateRequested if the node is not in CLEANWAIT state

Raises NoFreeConductorWorker when there is no free worker to start async task

Raises NodeLocked if node is locked by another conductor.

Raises NodeNotFound if the node no longer appears in the database

continue_node_deploy(*context*, *node_id*)

RPC method to continue deploying a node.

This is useful for deploying tasks that are async. When they complete, they call back via RPC, a new worker and lock are set up, and deploying continues. This can also be used to resume deploying on take_over.

Parameters

- **context** an admin context.
- **node_id** the ID or UUID of a node.

Raises InvalidStateRequested if the node is not in DEPLOYWAIT state

Raises NoFreeConductorWorker when there is no free worker to start async task

Raises NodeLocked if node is locked by another conductor.

Raises NodeNotFound if the node no longer appears in the database

create_allocation(***kwargs*)

create_node(***kwargs*)

create_port(***kwargs*)

destroy_allocation(***kwargs*)

destroy_node(***kwargs*)

destroy_port(***kwargs*)

destroy_portgroup(***kwargs*)

destroy_volume_connector(***kwargs*)

destroy_volume_target(***kwargs*)

do_node_clean(***kwargs*)

do_node_deploy(***kwargs*)

do_node_rescue(***kwargs*)

do_node_tear_down(***kwargs*)

do_node_unrescue(***kwargs*)

do_provisioning_action(***kwargs*)

driver_vendor_passthru(***kwargs*)

get_boot_device(***kwargs*)

get_console_information(***kwargs*)

get_driver_properties(***kwargs*)

get_driver_vendor_passthru_methods(***kwargs*)

get_indicator_state(***kwargs*)

get_node_vendor_passthru_methods(***kwargs*)

get_node_with_token(***kwargs*)

get_raid_logical_disk_properties(***kwargs*)

get_supported_boot_devices(***kwargs*)

get_supported_indicators(***kwargs*)

heartbeat(***kwargs*)

inject_nmi(***kwargs*)

inspect_hardware(***kwargs*)

manage_node_history(*context*)

object_action(*context*, *objinst*, *objmethod*, *args*, *kwargs*)
Perform an action on a VersionedObject instance.

Parameters

- **context** The context within which to perform the action
- **objinst** The object instance on which to perform the action
- **objmethod** The name of the action method to call
- **args** The positional arguments to the action method

- **kwargs** The keyword arguments to the action method

Returns A tuple with the updates made to the object and the result of the action method

object_backport_versions(*context*, *objinst*, *object_versions*)

Perform a backport of an object instance.

The default behavior of the base VersionedObjectSerializer, upon receiving an object with a version newer than what is in the local registry, is to call this method to request a backport of the object.

Parameters

- **context** The context within which to perform the backport
- **objinst** An instance of a VersionedObject to be backported
- **object_versions** A dict of {objname: version} mappings

Returns The downgraded instance of objinst

object_class_action_versions(*context*, *objname*, *objmethod*, *object_versions*, *args*, *kwargs*)

Perform an action on a VersionedObject class.

Parameters

- **context** The context within which to perform the action
- **objname** The registry name of the object
- **objmethod** The name of the action method to call
- **object_versions** A dict of {objname: version} mappings
- **args** The positional arguments to the action method
- **kwargs** The keyword arguments to the action method

Returns The result of the action method, which may (or may not) be an instance of the implementing VersionedObject class.

remove_node_traits(***kwargs*)

set_boot_device(***kwargs*)

set_console_mode(***kwargs*)

set_indicator_state(***kwargs*)

set_target_raid_config(***kwargs*)

target = <Target version=1.55>

update_node(***kwargs*)

update_port(***kwargs*)

update_portgroup(***kwargs*)

update_volume_connector(***kwargs*)

update_volume_target(***kwargs*)

validate_driver_interfaces(***kwargs*)

vendor_passthru(***kwargs*)

vif_attach(***kwargs*)

vif_detach(***kwargs*)

vif_list(***kwargs*)

ironic.conductor.manager.**do_sync_power_state**(*task*, *count*)
Sync the power state for this node, incrementing the counter on failure.

When the limit of power_state_sync_max_retries is reached, the node is put into maintenance mode and the error recorded.

Parameters

- **task** a TaskManager instance
- **count** number of times this node has previously failed a sync

Raises NodeLocked if unable to upgrade task lock to an exclusive one

Returns Count of failed attempts. On success, the counter is set to 0. On failure, the count is incremented by one

ironic.conductor.manager.**get_vendor_passthru_metadata**(*route_dict*)

ironic.conductor.manager.**handle_sync_power_state_max_retries_exceeded**(*task*, *actual_power_state*, *exception=None*)
Handles power state sync exceeding the max retries.

When synchronizing the power state between a node and the DB has exceeded the maximum number of retries, change the DB power state to be the actual node power state and place the node in maintenance.

Parameters

- **task** a TaskManager instance with an exclusive lock
- **actual_power_state** the actual power state of the node; a power state from ironic.common.states
- **exception** the exception object that caused the sync power state to fail, if present.

ironic.conductor.notification_utils module

ironic.conductor.notification_utils.**emit_console_notification**(*task*, *action*, *status*)
Helper for conductor sending a set console state notification.

Parameters

- **task** a TaskManager instance.
- **action** Action string to go in the EventType. Must be either console_set or console_restore.
- **status** One of *ironic.objects.fields.NotificationStatus.START*, END or ERROR.

ironic.conductor.notification_utils.**emit_power_set_notification**(*task*, *level*, *status*, *to_power*)

Helper for conductor sending a set power state notification.

Parameters

- **task** a TaskManager instance.
- **level** Notification level. One of *ironic.objects.fields.NotificationLevel.ALL*
- **status** Status to go in the EventType. One of *ironic.objects.fields.NotificationStatus.SUCCESS* or ERROR. ERROR indicates that ironic-conductor couldnt retrieve the power state for this node, or that it couldnt set the power state of the node.
- **to_power** the power state the conductor is attempting to set on the node. This is used instead of the nodes target_power_state attribute since the baremetal.node.power_set.start notification is sent early, before target_power_state is set on the node.

ironic.conductor.notification_utils.**emit_power_state_corrected_notification**(*task*, *from_power*)

Helper for conductor sending a node power state corrected notification.

When ironic detects that the actual power state on a bare metal hardware is different from the power state on an ironic node (DB), the ironic nodes power state is corrected to be that of the bare metal hardware. A notification is emitted about this after the database is updated to reflect this correction.

Parameters

- **task** a TaskManager instance.
- **from_power** the power state of the node before this change was detected

ironic.conductor.notification_utils.**emit_provision_set_notification**(*task*, *level*, *status*, *prev_state*, *prev_target*, *event*)

Helper for conductor sending a set provision state notification.

Parameters

- **task** a TaskManager instance.
- **level** One of fields.NotificationLevel.
- **status** One of fields.NotificationStatus.
- **prev_state** Previous provision state.
- **prev_target** Previous target provision state.
- **event** FSM event that triggered provision state change.

ironic.conductor.periodics module

Conductor periodics.

exception ironic.conductor.periodics.**Stop**
Bases: Exception

A signal to stop the current iteration of a periodic task.

ironic.conductor.periodics.**node_periodic**(*purpose*, *spacing*, *enabled=True*, *filters=None*, *predicate=None*, *predicate_extra_fields=()*, *limit=None*, *shared_task=True*)
A decorator to define a periodic task to act on nodes.

Defines a periodic task that fetches the list of nodes mapped to the current conductor which satisfy the provided filters.

The decorated function must be a method on either the conductor manager or a hardware interface. The signature is:

- for conductor manager: `(self, task, context)`
- for hardware interfaces: `(self, task, manager, context)`.

When the periodic is running on a hardware interface, only tasks using this interface are considered.

`NodeNotFound` and `NodeLocked` exceptions are ignored. Raise `Stop` to abort the current iteration of the task and reschedule it.

Parameters

- **purpose** a human-readable description of the activity, e.g. verifying that the cat is purring.
- **spacing** how often (in seconds) to run the periodic task.
- **enabled** whether the task is enabled; defaults to `spacing > 0`.
- **filters** database-level filters for the nodes.
- **predicate** a callable to run on the fetched nodes *before* creating a task for them. The only parameter will be a named tuple with fields `uuid`, `driver`, `conductor_group` plus everything from `predicate_extra_fields`. If the callable accepts a 2nd parameter, it will be the conductor manager instance.
- **predicate_extra_fields** extra fields to fetch on the initial request and pass into the `predicate`. Must not contain `uuid`, `driver` and `conductor_group` since they are always included.
- **limit** how many nodes to process before stopping the current iteration. If `predicate` returns `False`, the node is not counted. If the decorated function returns `False`, the node is not counted either. Can be a callable, in which case it will be called on each iteration to determine the limit.
- **shared_task** if `True`, the task will have a shared lock. It is recommended to start with a shared lock and upgrade it only if needed.

ironic.conductor.periodics.**periodic**(*spacing*, *enabled=True*, ***kwargs*)
A decorator to define a periodic task.

Parameters

- **spacing** how often (in seconds) to run the periodic task.
- **enabled** whether the task is enabled; defaults to `spacing > 0`.

ironic.conductor.rpcapi module

Client side of the conductor RPC API.

class ironic.conductor.rpcapi.**ConductorAPI**(*topic=None*)

Bases: `object`

Client side of the conductor RPC API.

API version history:

1.0 - Initial version.
    Included get_node_power_status
1.1 - Added update_node and start_power_state_change.
1.2 - Added vendor_passthru.
1.3 - Rename start_power_state_change to change_node_power_state.
1.4 - Added do_node_deploy and do_node_tear_down.
1.5 - Added validate_driver_interfaces.
1.6 - change_node_power_state, do_node_deploy and do_node_tear_down
    accept node id instead of node object.
1.7 - Added topic parameter to RPC methods.
1.8 - Added change_node_maintenance_mode.
1.9 - Added destroy_node.
1.10 - Remove get_node_power_state
1.11 - Added get_console_information, set_console_mode.
1.12 - validate_vendor_action, do_vendor_action replaced by single
    vendor_passthru method.
1.13 - Added update_port.
1.14 - Added driver_vendor_passthru.
1.15 - Added rebuild parameter to do_node_deploy.
1.16 - Added get_driver_properties.
1.17 - Added set_boot_device, get_boot_device and
    get_supported_boot_devices.
1.18 - Remove change_node_maintenance_mode.
1.19 - Change return value of vendor_passthru and
    driver_vendor_passthru
1.20 - Added http_method parameter to vendor_passthru and
    driver_vendor_passthru
1.21 - Added get_node_vendor_passthru_methods and
    get_driver_vendor_passthru_methods
1.22 - Added configdrive parameter to do_node_deploy.
1.23 - Added do_provisioning_action
1.24 - Added inspect_hardware method

1.25 - Added destroy_port
1.26 - Added continue_node_clean
1.27 - Convert continue_node_clean to cast
1.28 - Change exceptions raised by destroy_node
1.29 - Change return value of vendor_passthru and
driver_vendor_passthru to a dictionary
1.30 - Added set_target_raid_config and
get_raid_logical_disk_properties
1.31 - Added Versioned Objects indirection API methods:
object_class_action_versions, object_action and
object_backport_versions
1.32 - Add do_node_clean
1.33 - Added update and destroy portgroup.
1.34 - Added heartbeat
1.35 - Added destroy_volume_connector and update_volume_connector
1.36 - Added create_node
1.37 - Added destroy_volume_target and update_volume_target
1.38 - Added vif_attach, vif_detach, vif_list
1.39 - Added timeout optional parameter to change_node_power_state
1.40 - Added inject_nmi
1.41 - Added create_port
1.42 - Added optional agent_version to heartbeat
1.43 - Added do_node_rescue, do_node_unrescue and can_send_rescue
1.44 - Added add_node_traits and remove_node_traits.
1.45 - Added continue_node_deploy
1.46 - Added reset_interfaces to update_node
1.47 - Added support for conductor groups
1.48 - Added allocation API
1.49 - Added get_node_with_token and agent_token argument to heartbeat
1.50 - Added set_indicator_state, get_indicator_state and
get_supported_indicators.
1.51 - Added agent_verify_ca to heartbeat.
1.52 - Added deploy steps argument to provisioning
1.53 - Added disable_ramdisk to do_node_clean.
1.54 - Added optional agent_status and agent_status_message to heartbeat
1.55 - Added change_node_boot_mode

RPC_API_VERSION = '1.55'

add_node_traits(*context*, *node_id*, *traits*, *replace=False*, *topic=None*)
Add or replace traits for a node.

Parameters

- **context** request context.
- **node_id** node ID or UUID.

- **traits** a list of traits to add to the node.
- **replace** True to replace all of the nodes traits.
- **topic** RPC topic. Defaults to self.topic.

Raises InvalidParameterValue if adding the traits would exceed the per-node traits limit.

Raises NodeLocked if node is locked by another conductor.

Raises NodeNotFound if the node does not exist.

can_send_create_port()
Return whether the RPCAPI supports the create_port method.

can_send_rescue()
Return whether the RPCAPI supports node rescue methods.

change_node_boot_mode(*context*, *node_id*, *new_state*, *topic=None*)
Change a nodes boot mode.

Synchronously, acquire lock and start the conductor background task to change boot mode of a node.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **new_state** one of ironic.common.boot_modes values (bios or uefi)
- **topic** RPC topic. Defaults to self.topic.

Raises NoFreeConductorWorker when there is no free worker to start async task.

change_node_power_state(*context*, *node_id*, *new_state*, *topic=None*, *timeout=None*)
Change a nodes power state.

Synchronously, acquire lock and start the conductor background task to change power state of a node.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **new_state** one of ironic.common.states power state values
- **timeout** timeout (in seconds) positive integer (> 0) for any power state. None indicates to use default timeout.
- **topic** RPC topic. Defaults to self.topic.

Raises NoFreeConductorWorker when there is no free worker to start async task.

change_node_secure_boot(*context*, *node_id*, *new_state*, *topic=None*)
Change a nodes secure_boot state.

Synchronously, acquire lock and start the conductor background task to change secure_boot state of a node.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **new_state** Target secure boot state (True => on or False => off)
- **topic** RPC topic. Defaults to self.topic.

Raises NoFreeConductorWorker when there is no free worker to start async task.

continue_node_clean(*context*, *node_id*, *topic=None*)
Signal to conductor service to start the next cleaning action.

NOTE(JoshNang) this is an RPC cast, there will be no response or exception raised by the conductor for this RPC.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **topic** RPC topic. Defaults to self.topic.

continue_node_deploy(*context*, *node_id*, *topic=None*)
Signal to conductor service to start the next deployment action.

NOTE(rloo): this is an RPC cast, there will be no response or exception raised by the conductor for this RPC.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **topic** RPC topic. Defaults to self.topic.

create_allocation(*context*, *allocation*, *topic=None*)
Create an allocation.

Parameters

- **context** request context.
- **allocation** an allocation object.
- **topic** RPC topic. Defaults to self.topic.

create_node(*context*, *node_obj*, *topic=None*)
Synchronously, have a conductor validate and create a node.

Create the nodes information in the database and return a node object.

Parameters

- **context** request context.
- **node_obj** a created (but not saved) node object.
- **topic** RPC topic. Defaults to self.topic.

Returns created node object.

Raises InterfaceNotFoundInEntrypoint if validation fails for any dynamic interfaces (e.g. network_interface).

Raises NoValidDefaultForInterface if no default can be calculated for some interfaces, and explicit values must be provided.

`create_port`(*context*, *port_obj*, *topic=None*)
Synchronously, have a conductor validate and create a port.

Create the ports information in the database and return a port object. The conductor will lock related node and trigger specific driver actions if they are needed.

Parameters

- `context` request context.
- `port_obj` a created (but not saved) port object.
- `topic` RPC topic. Defaults to self.topic.

Returns created port object.

`destroy_allocation`(*context*, *allocation*, *topic=None*)
Delete an allocation.

Parameters

- `context` request context.
- `allocation` an allocation object.
- `topic` RPC topic. Defaults to self.topic.

Raises InvalidState if the associated node is in the wrong provision state to perform deallocation.

`destroy_node`(*context*, *node_id*, *topic=None*)
Delete a node.

Parameters

- `context` request context.
- `node_id` node id or uuid.
- `topic` RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor.

Raises NodeAssociated if the node contains an instance associated with it.

Raises InvalidState if the node is in the wrong provision state to perform deletion.

`destroy_port`(*context*, *port*, *topic=None*)
Delete a port.

Parameters

- `context` request context.
- `port` port object
- `topic` RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor.

Raises NodeNotFound if the node associated with the port does not exist.

destroy_portgroup(*context*, *portgroup*, *topic=None*)
Delete a portgroup.

Parameters

- **context** request context.
- **portgroup** portgroup object
- **topic** RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor.

Raises NodeNotFound if the node associated with the portgroup does not exist.

Raises PortgroupNotEmpty if portgroup is not empty

destroy_volume_connector(*context*, *connector*, *topic=None*)
Delete a volume connector.

Delete the volume connector. The conductor will lock the related node during this operation.

Parameters

- **context** request context
- **connector** volume connector object
- **topic** RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor

Raises NodeNotFound if the node associated with the connector does not exist

Raises VolumeConnectorNotFound if the volume connector cannot be found

destroy_volume_target(*context*, *target*, *topic=None*)
Delete a volume target.

Parameters

- **context** request context
- **target** volume target object
- **topic** RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor

Raises NodeNotFound if the node associated with the target does not exist

Raises VolumeTargetNotFound if the volume target cannot be found

do_node_clean(*context*, *node_id*, *clean_steps*, *disable_ramdisk=None*, *topic=None*)
Signal to conductor service to perform manual cleaning on a node.

Parameters

- **context** request context.
- **node_id** node ID or UUID.
- **clean_steps** a list of clean step dictionaries.
- **disable_ramdisk** Whether to skip booting ramdisk for cleaning.
- **topic** RPC topic. Defaults to self.topic.

Raises InvalidParameterValue if validation of power driver interface failed.

Raises InvalidStateRequested if cleaning can not be performed.

Raises NodeInMaintenance if node is in maintenance mode.

Raises NodeLocked if node is locked by another conductor.

Raises NoFreeConductorWorker when there is no free worker to start async task.

`do_node_deploy`(*context*, *node_id*, *rebuild*, *configdrive*, *topic=None*, *deploy_steps=None*)
Signal to conductor service to perform a deployment.

Parameters

- `context` request context.
- `node_id` node id or uuid.
- `rebuild` True if this is a rebuild request.
- `configdrive` A gzipped and base64 encoded configdrive.
- `topic` RPC topic. Defaults to self.topic.
- `deploy_steps` Deploy steps

Raises InstanceDeployFailure

Raises InvalidParameterValue if validation fails

Raises MissingParameterValue if a required parameter is missing

Raises NoFreeConductorWorker when there is no free worker to start async task.

The node must already be configured and in the appropriate undeployed state before this method is called.

`do_node_rescue`(*context*, *node_id*, *rescue_password*, *topic=None*)
Signal to conductor service to perform a rescue.

Parameters

- `context` request context.
- `node_id` node ID or UUID.
- `rescue_password` A string representing the password to be set inside the rescue environment.
- `topic` RPC topic. Defaults to self.topic.

Raises InstanceRescueFailure

Raises NoFreeConductorWorker when there is no free worker to start async task.

The node must already be configured and in the appropriate state before this method is called.

`do_node_tear_down`(*context*, *node_id*, *topic=None*)
Signal to conductor service to tear down a deployment.

Parameters

- `context` request context.
- `node_id` node id or uuid.

- **topic** RPC topic. Defaults to self.topic.

Raises InstanceDeployFailure

Raises InvalidParameterValue if validation fails

Raises MissingParameterValue if a required parameter is missing

Raises NoFreeConductorWorker when there is no free worker to start async task.

The node must already be configured and in the appropriate deployed state before this method is called.

do_node_unrescue(*context*, *node_id*, *topic=None*)
Signal to conductor service to perform an unrescue.

Parameters

- **context** request context.
- **node_id** node ID or UUID.
- **topic** RPC topic. Defaults to self.topic.

Raises InstanceUnrescueFailure

Raises NoFreeConductorWorker when there is no free worker to start async task.

The node must already be configured and in the appropriate state before this method is called.

do_provisioning_action(*context*, *node_id*, *action*, *topic=None*)
Signal to conductor service to perform the given action on a node.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **action** an action. One of ironic.common.states.VERBS
- **topic** RPC topic. Defaults to self.topic.

Raises InvalidParameterValue

Raises NoFreeConductorWorker when there is no free worker to start async task.

Raises InvalidStateRequested if the requested action can not be performed.

This encapsulates some provisioning actions in a single call.

driver_vendor_passthru(*context*, *driver_name*, *driver_method*, *http_method*, *info*, *topic=None*)
Pass vendor-specific calls which dont specify a node to a driver.

Handles driver-level vendor passthru calls. These calls dont require a node UUID and are executed on a random conductor with the specified driver. If the method mode is async the conductor will start background worker to perform vendor action.

Parameters

- **context** request context.
- **driver_name** name of the driver on which to call the method.
- **driver_method** name of the vendor method, for use by the driver.

- **http_method** the HTTP method used for the request.
- **info** data to pass through to the driver.
- **topic** RPC topic. Defaults to self.topic.

Raises InvalidParameterValue for parameter errors.

Raises MissingParameterValue if a required parameter is missing

Raises UnsupportedDriverExtension if the driver doesnt have a vendor interface, or if the vendor interface does not support the specified driver_method.

Raises DriverNotFound if the supplied driver is not loaded.

Raises NoFreeConductorWorker when there is no free worker to start async task.

Raises InterfaceNotFoundInEntrypoint if the default interface for a hardware type is invalid.

Raises NoValidDefaultForInterface if no default interface implementation can be found for this drivers vendor interface.

Returns

A dictionary containing:

return The response of the invoked vendor method

async Boolean value. Whether the method was invoked asynchronously (True) or synchronously (False). When invoked asynchronously the response will be always None.

attach Boolean value. Whether to attach the response of the invoked vendor method to the HTTP response object (True) or return it in the response body (False).

get_boot_device(*context*, *node_id*, *topic=None*)

Get the current boot device.

Returns the current boot device of a node.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **topic** RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor.

Raises UnsupportedDriverExtension if the nodes driver doesnt support management.

Raises InvalidParameterValue when the wrong driver info is specified.

Raises MissingParameterValue if missing supplied info.

Returns

a dictionary containing:

boot_device the boot device, one of *ironic.common.boot_devices* or None if it is unknown.

persistent Whether the boot device will persist to all future boots or not, None if it is unknown.

get_conductor_for(*node*)
Get the conductor which the node is mapped to.

Parameters **node** a node object.

Returns the conductor hostname.

Raises NoValidHost

get_console_information(*context*, *node_id*, *topic=None*)
Get connection information about the console.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **topic** RPC topic. Defaults to self.topic.

Raises UnsupportedDriverExtension if the nodes driver doesnt support console.

Raises InvalidParameterValue when the wrong driver info is specified.

Raises MissingParameterValue if a required parameter is missing

get_current_topic()
Get RPC topic name for the current conductor.

get_driver_properties(*context*, *driver_name*, *topic=None*)
Get the properties of the driver.

Parameters

- **context** request context.
- **driver_name** name of the driver.
- **topic** RPC topic. Defaults to self.topic.

Returns a dictionary with <property name>:<property description> entries.

Raises DriverNotFound.

get_driver_vendor_passthru_methods(*context*, *driver_name*, *topic=None*)
Retrieve information about vendor methods of the given driver.

Parameters

- **context** an admin context.
- **driver_name** name of the driver.
- **topic** RPC topic. Defaults to self.topic.

Raises UnsupportedDriverExtension if current driver does not have vendor interface.

Raises DriverNotFound if the supplied driver is not loaded.

Raises InterfaceNotFoundInEntrypoint if the default interface for a hardware type is invalid.

Raises NoValidDefaultForInterface if no default interface implementation can be found for this drivers vendor interface.

Returns dictionary of <method name>:<method metadata> entries.

`get_indicator_state`(*context*, *node_id*, *component*, *indicator*, *topic=None*)
Get node hardware component indicator state.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **component** The hardware component, one of *ironic.common.components*.
- **indicator** Indicator IDs, as reported by *get_supported_indicators*)
- **topic** RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor.

Raises UnsupportedDriverExtension if the nodes driver doesnt support management.

Raises InvalidParameterValue when the wrong driver info is specified.

Raises MissingParameterValue if missing supplied info.

Returns Indicator state, one of mod:*ironic.common.indicator_states*.

`get_node_vendor_passthru_methods`(*context*, *node_id*, *topic=None*)
Retrieve information about vendor methods of the given node.

Parameters

- **context** an admin context.
- **node_id** the id or uuid of a node.
- **topic** RPC topic. Defaults to self.topic.

Returns dictionary of <method name>:<method metadata> entries.

`get_node_with_token`(*context*, *node_id*, *topic=None*)
Request the node from the conductor with an agent token

Parameters

- **context** request context.
- **node_id** node ID or UUID.
- **topic** RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor.

Returns A Node object with agent token.

`get_raid_logical_disk_properties`(*context*, *driver_name*, *topic=None*)
Get the logical disk properties for RAID configuration.

Gets the information about logical disk properties which can be specified in the input RAID configuration.

Parameters

- **context** request context.
- **driver_name** name of the driver
- **topic** RPC topic. Defaults to self.topic.

Raises UnsupportedDriverExtension if the driver doesnt support RAID configuration.

Raises InterfaceNotFoundInEntrypoint if the default interface for a hardware type is invalid.

Raises NoValidDefaultForInterface if no default interface implementation can be found for this drivers RAID interface.

Returns A dictionary containing the properties that can be mentioned for logical disks and a textual description for them.

get_random_topic()
Get an RPC topic for a random conductor service.

get_supported_boot_devices(*context*, *node_id*, *topic=None*)
Get the list of supported devices.

Returns the list of supported boot devices of a node.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **topic** RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor.

Raises UnsupportedDriverExtension if the nodes driver doesnt support management.

Raises InvalidParameterValue when the wrong driver info is specified.

Raises MissingParameterValue if missing supplied info.

Returns A list with the supported boot devices defined in *ironic.common.boot_devices*.

get_supported_indicators(*context*, *node_id*, *component=None*, *topic=None*)
Get node hardware components and their indicators.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **component** The hardware component, one of *ironic.common.components*.
- **topic** RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor.

Raises UnsupportedDriverExtension if the nodes driver doesnt support management.

Raises InvalidParameterValue when the wrong driver info is specified.

Raises MissingParameterValue if missing supplied info.

Returns

A dictionary of hardware components (*ironic.common.components*) as keys with indicator IDs as values.

```
{
    'chassis': ['enclosure-0'],
    'system': ['blade-A']
    'drive': ['ssd0']
}
```

get_topic_for(*node*)

Get the RPC topic for the conductor service the node is mapped to.

Parameters **node** a node object.

Returns an RPC topic string.

Raises NoValidHost

get_topic_for_driver(*driver_name*)

Get RPC topic name for a conductor supporting the given driver.

The topic is used to route messages to the conductor supporting the specified driver. A conductor is selected at random from the set of qualified conductors.

Parameters **driver_name** the name of the driver to route to.

Returns an RPC topic string.

Raises DriverNotFound

heartbeat(*context*, *node_id*, *callback_url*, *agent_version*, *agent_token=None*, *agent_verify_ca=None*, *agent_status=None*, *agent_status_message=None*, *topic=None*)

Process a node heartbeat.

Parameters

- **context** request context.
- **node_id** node ID or UUID.
- **callback_url** URL to reach back to the ramdisk.
- **topic** RPC topic. Defaults to self.topic.
- **agent_token** randomly generated validation token.
- **agent_version** the version of the agent that is heartbeating
- **agent_verify_ca** TLS certificate for the agent.
- **agent_status** The status of the agent that is heartbeating
- **agent_status_message** Optional message describing the agent status

Raises InvalidParameterValue if an invalid agent token is received.

inject_nmi(*context*, *node_id*, *topic=None*)
Inject NMI for a node.

Inject NMI (Non Maskable Interrupt) for a node immediately. Be aware that not all drivers support this.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **topic** RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor.

Raises UnsupportedDriverExtension if the nodes driver doesnt support management or management.inject_nmi.

Raises InvalidParameterValue when the wrong driver info is specified or an invalid boot device is specified.

Raises MissingParameterValue if missing supplied info.

inspect_hardware(*context*, *node_id*, *topic=None*)
Signals the conductor service to perform hardware introspection.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **topic** RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor.

Raises HardwareInspectionFailure

Raises NoFreeConductorWorker when there is no free worker to start async task.

Raises UnsupportedDriverExtension if the nodes driver doesnt support inspection.

Raises InvalidStateRequested if inspect is not a valid action to do in the current state.

object_action(*context*, *objinst*, *objmethod*, *args*, *kwargs*)
Perform an action on a VersionedObject instance.

We want any conductor to handle this, so it is intentional that there is no topic argument for this method.

Parameters

- **context** The context within which to perform the action
- **objinst** The object instance on which to perform the action
- **objmethod** The name of the action method to call
- **args** The positional arguments to the action method
- **kwargs** The keyword arguments to the action method

Raises NotImplementedError when an operator makes an error during upgrade

Returns A tuple with the updates made to the object and the result of the action method

`object_backport_versions`(*context*, *objinst*, *object_versions*)
Perform a backport of an object instance.

The default behavior of the base VersionedObjectSerializer, upon receiving an object with a version newer than what is in the local registry, is to call this method to request a backport of the object.

We want any conductor to handle this, so it is intentional that there is no topic argument for this method.

Parameters

- **context** The context within which to perform the backport
- **objinst** An instance of a VersionedObject to be backported
- **object_versions** A dict of {objname: version} mappings

Raises NotImplementedError when an operator makes an error during upgrade

Returns The downgraded instance of objinst

`object_class_action_versions`(*context*, *objname*, *objmethod*, *object_versions*, *args*, *kwargs*)
Perform an action on a VersionedObject class.

We want any conductor to handle this, so it is intentional that there is no topic argument for this method.

Parameters

- **context** The context within which to perform the action
- **objname** The registry name of the object
- **objmethod** The name of the action method to call
- **object_versions** A dict of {objname: version} mappings
- **args** The positional arguments to the action method
- **kwargs** The keyword arguments to the action method

Raises NotImplementedError when an operator makes an error during upgrade

Returns The result of the action method, which may (or may not) be an instance of the implementing VersionedObject class.

`remove_node_traits`(*context*, *node_id*, *traits*, *topic=None*)
Remove some or all traits from a node.

Parameters

- **context** request context.
- **node_id** node ID or UUID.
- **traits** a list of traits to remove from the node, or None. If None, all traits will be removed from the node.

- **topic** RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor.

Raises NodeNotFound if the node does not exist.

Raises NodeTraitNotFound if one of the traits is not found.

set_boot_device(*context*, *node_id*, *device*, *persistent=False*, *topic=None*)
Set the boot device for a node.

Set the boot device to use on next reboot of the node. Be aware that not all drivers support this.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **device** the boot device, one of *ironic.common.boot_devices*.
- **persistent** Whether to set next-boot, or make the change permanent. Default: False.
- **topic** RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor.

Raises UnsupportedDriverExtension if the nodes driver doesnt support management.

Raises InvalidParameterValue when the wrong driver info is specified or an invalid boot device is specified.

Raises MissingParameterValue if missing supplied info.

set_console_mode(*context*, *node_id*, *enabled*, *topic=None*)
Enable/Disable the console.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **topic** RPC topic. Defaults to self.topic.
- **enabled** Boolean value; whether the console is enabled or disabled.

Raises UnsupportedDriverExtension if the nodes driver doesnt support console.

Raises InvalidParameterValue when the wrong driver info is specified.

Raises MissingParameterValue if a required parameter is missing

Raises NoFreeConductorWorker when there is no free worker to start async task.

set_indicator_state(*context*, *node_id*, *component*, *indicator*, *state*, *topic=None*)
Set node hardware components indicator to the desired state.

Parameters

- **context** request context.
- **node_id** node id or uuid.

- **component** The hardware component, one of *ironic.common.components*.
- **indicator** Indicator IDs, as reported by *get_supported_indicators*)
- **state** Indicator state, one of mod:*ironic.common.indicator_states*.
- **topic** RPC topic. Defaults to self.topic.

Raises NodeLocked if node is locked by another conductor.

Raises UnsupportedDriverExtension if the nodes driver doesnt support management.

Raises InvalidParameterValue when the wrong driver info is specified or an invalid boot device is specified.

Raises MissingParameterValue if missing supplied info.

set_target_raid_config(*context*, *node_id*, *target_raid_config*, *topic=None*)

Stores the target RAID configuration on the node.

Stores the target RAID configuration on node.target_raid_config

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **target_raid_config** Dictionary containing the target RAID configuration. It may be an empty dictionary as well.
- **topic** RPC topic. Defaults to self.topic.

Raises UnsupportedDriverExtension if the nodes driver doesnt support RAID configuration.

Raises InvalidParameterValue, if validation of target raid config fails.

Raises MissingParameterValue, if some required parameters are missing.

Raises NodeLocked if node is locked by another conductor.

update_node(*context*, *node_obj*, *topic=None*, *reset_interfaces=False*)

Synchronously, have a conductor update the nodes information.

Update the nodes information in the database and return a node object. The conductor will lock the node while it validates the supplied information. If driver_info is passed, it will be validated by the core drivers. If instance_uuid is passed, it will be set or unset only if the node is properly configured.

Note that power_state should not be passed via this method. Use change_node_power_state for initiating driver actions.

Parameters

- **context** request context.
- **node_obj** a changed (but not saved) node object.
- **topic** RPC topic. Defaults to self.topic.
- **reset_interfaces** whether to reset hardware interfaces to their defaults.

Returns updated node object, including all fields.

Raises NoValidDefaultForInterface if no default can be calculated for some interfaces, and explicit values must be provided.

`update_port`(*context*, *port_obj*, *topic=None*)
Synchronously, have a conductor update the ports information.

Update the ports information in the database and return a port object. The conductor will lock related node and trigger specific driver actions if they are needed.

Parameters

- `context` request context.
- `port_obj` a changed (but not saved) port object.
- `topic` RPC topic. Defaults to self.topic.

Returns updated port object, including all fields.

`update_portgroup`(*context*, *portgroup_obj*, *topic=None*)
Synchronously, have a conductor update the portgroups information.

Update the portgroups information in the database and return a portgroup object. The conductor will lock related node and trigger specific driver actions if they are needed.

Parameters

- `context` request context.
- `portgroup_obj` a changed (but not saved) portgroup object.
- `topic` RPC topic. Defaults to self.topic.

Returns updated portgroup object, including all fields.

`update_volume_connector`(*context*, *connector*, *topic=None*)
Update the volume connectors information.

Update the volume connectors information in the database and return a volume connector object. The conductor will lock the related node during this operation.

Parameters

- `context` request context
- `connector` a changed (but not saved) volume connector object
- `topic` RPC topic. Defaults to self.topic.

Raises InvalidParameterValue if the volume connectors UUID is being changed

Raises NodeLocked if node is locked by another conductor

Raises NodeNotFound if the node associated with the connector does not exist

Raises VolumeConnectorNotFound if the volume connector cannot be found

Raises VolumeConnectorTypeAndIdAlreadyExists if another connector already exists with the same values for type and connector_id fields

Returns updated volume connector object, including all fields.

update_volume_target(*context*, *target*, *topic=None*)
Update the volume targets information.

Update the volume targets information in the database and return a volume target object. The conductor will lock the related node during this operation.

Parameters

- **context** request context
- **target** a changed (but not saved) volume target object
- **topic** RPC topic. Defaults to self.topic.

Raises InvalidParameterValue if the volume targets UUID is being changed

Raises NodeLocked if the node is already locked

Raises NodeNotFound if the node associated with the volume target does not exist

Raises VolumeTargetNotFound if the volume target cannot be found

Raises VolumeTargetBootIndexAlreadyExists if a volume target already exists with the same node ID and boot index values

Returns updated volume target object, including all fields

validate_driver_interfaces(*context*, *node_id*, *topic=None*)
Validate the *core* and *standardized* interfaces for drivers.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **topic** RPC topic. Defaults to self.topic.

Returns a dictionary containing the results of each interface validation.

vendor_passthru(*context*, *node_id*, *driver_method*, *http_method*, *info*, *topic=None*)
Receive requests for vendor-specific actions.

Synchronously validate driver specific info or get driver status, and if successful invokes the vendor method. If the method mode is async the conductor will start background worker to perform vendor action.

Parameters

- **context** request context.
- **node_id** node id or uuid.
- **driver_method** name of method for driver.
- **http_method** the HTTP method used for the request.
- **info** info for node driver.
- **topic** RPC topic. Defaults to self.topic.

Raises InvalidParameterValue if supplied info is not valid.

Raises MissingParameterValue if a required parameter is missing

Raises UnsupportedDriverExtension if current driver does not have vendor interface.

Raises NoFreeConductorWorker when there is no free worker to start async task.

Raises NodeLocked if node is locked by another conductor.

Returns

A dictionary containing:

return The response of the invoked vendor method

async Boolean value. Whether the method was invoked asynchronously (True) or synchronously (False). When invoked asynchronously the response will be always None.

attach Boolean value. Whether to attach the response of the invoked vendor method to the HTTP response object (True) or return it in the response body (False).

`vif_attach`(*context*, *node_id*, *vif_info*, *topic=None*)
Attach VIF to a node

Parameters

- `context` request context.
- `node_id` node ID or UUID.
- `vif_info` a dictionary representing VIF object. It must have an id key, whose value is a unique identifier for that VIF.
- `topic` RPC topic. Defaults to self.topic.

Raises NodeLocked, if node has an exclusive lock held on it

Raises NetworkError, if an error occurs during attaching the VIF.

Raises InvalidParameterValue, if a parameter thats required for VIF attach is wrong/missing.

`vif_detach`(*context*, *node_id*, *vif_id*, *topic=None*)
Detach VIF from a node

Parameters

- `context` request context.
- `node_id` node ID or UUID.
- `vif_id` an ID of a VIF.
- `topic` RPC topic. Defaults to self.topic.

Raises NodeLocked, if node has an exclusive lock held on it

Raises NetworkError, if an error occurs during detaching the VIF.

Raises InvalidParameterValue, if a parameter thats required for VIF detach is wrong/missing.

`vif_list`(*context*, *node_id*, *topic=None*)
List attached VIFs for a node

Parameters

- **context** request context.
- **node_id** node ID or UUID.
- **topic** RPC topic. Defaults to self.topic.

Returns List of VIF dictionaries, each dictionary will have an id entry with the ID of the VIF.

Raises NetworkError, if an error occurs during listing the VIFs.

Raises InvalidParameterValue, if a parameter thats required for VIF list is wrong/missing.

class ironic.conductor.rpcapi.**LocalContext**

Bases: object

Context to make calls to a local conductor.

call(*context*, *rpc_call_name*, ***kwargs*)

Make a local conductor call.

cast(*context*, *rpc_call_name*, ***kwargs*)

Make a local conductor call.

It is expected that the underlying call uses a thread to avoid blocking the caller.

Any exceptions are logged and ignored.

ironic.conductor.steps module

ironic.conductor.steps.**find_step**(*steps*, *step*)

Find an identical step in the list of steps.

ironic.conductor.steps.**is_equivalent**(*step1*, *step2*)

Compare steps, ignoring their priority.

ironic.conductor.steps.**set_node_cleaning_steps**(*task*, *disable_ramdisk=False*)

Set up the node with clean step information for cleaning.

For automated cleaning, get the clean steps from the driver. For manual cleaning, the users clean steps are known but need to be validated against the drivers clean steps.

Parameters **disable_ramdisk** If *True*, only steps with requires_ramdisk=False are accepted.

Raises InvalidParameterValue if there is a problem with the users clean steps.

Raises NodeCleaningFailure if there was a problem getting the clean steps.

ironic.conductor.steps.**set_node_deployment_steps**(*task*, *reset_current=True*, *skip_missing=False*)

Set up the node with deployment step information for deploying.

Get the deploy steps from the driver.

Parameters **reset_current** Whether to reset the current step to the first one.

Raises InstanceDeployFailure if there was a problem getting the deployment steps.

ironic.conductor.steps.**step_id**(*step*)

Return the ID of a deploy step.

The ID is a string, <interface>.<step>.

Parameters **step** the step dictionary.

Returns the steps ID string.

ironic.conductor.steps.**validate_user_deploy_steps_and_templates**(*task*, *deploy_steps=None*, *skip_missing=False*)

Validate the user deploy steps and the deploy templates for a node.

Parameters

- **task** A TaskManager object
- **deploy_steps** Deploy steps to validate. Optional. If not provided then will check nodes driver internal info.
- **skip_missing** whether skip missing steps that are not yet available at the time of validation.

Raises InvalidParameterValue if the instance has traits that map to deploy steps that are unsupported by the nodes driver interfaces or user deploy steps are unsupported by the nodes driver interfaces

Raises InstanceDeployFailure if there was a problem getting the deploy steps from the driver.

ironic.conductor.task_manager module

A context manager to perform a series of tasks on a set of resources.

TaskManager is a context manager, created on-demand to allow synchronized access to a node and its resources.

The *TaskManager* will, by default, acquire an exclusive lock on a node for the duration that the TaskManager instance exists. You may create a TaskManager instance without locking by passing shared=True when creating it, but certain operations on the resources held by such an instance of TaskManager will not be possible. Requiring this exclusive lock guards against parallel operations interfering with each other.

A shared lock is useful when performing non-interfering operations, such as validating the driver interfaces.

An exclusive lock is stored in the database to coordinate between *ironic.conductor.manager* instances, that are typically deployed on different hosts.

TaskManager methods, as well as driver methods, may be decorated to determine whether their invocation requires an exclusive lock.

The TaskManager instance exposes certain node resources and properties as attributes that you may access:

task.context The context passed to TaskManager()

task.shared False if Node is locked, True if it is not locked. (The shared kwarg arg of TaskManager())

task.node The Node object

task.ports Ports belonging to the Node

task.portgroups Portgroups belonging to the Node

task.volume_connectors Storage connectors belonging to the Node

task.volume_targets Storage targets assigned to the Node

task.driver The Driver for the Node, or the Driver based on the driver_name kwarg of TaskManager().

Example usage:

```
with task_manager.acquire(context, node_id, purpose='power on') as task:
    task.driver.power.power_on(task.node)
```

If you need to execute task-requiring code in a background thread, the TaskManager instance provides an interface to handle this for you, making sure to release resources when the thread finishes (successfully or if an exception occurs). Common use of this is within the Manager like so:

```
with task_manager.acquire(context, node_id, purpose='some work') as task:
    <do some work>
    task.spawn_after(self._spawn_worker,
                     utils.node_power_action, task, new_state)
```

All exceptions that occur in the current GreenThread as part of the spawn handling are re-raised. You can specify a hook to execute custom code when such exceptions occur. For example, the hook is a more elegant solution than wrapping the with task_manager.acquire() with a try..exception block. (Note that this hook does not handle exceptions raised in the background thread.):

```
def on_error(e):
    if isinstance(e, Exception):
        ...

with task_manager.acquire(context, node_id, purpose='some work') as task:
    <do some work>
    task.set_spawn_error_hook(on_error)
    task.spawn_after(self._spawn_worker,
                     utils.node_power_action, task, new_state)
```

class ironic.conductor.task_manager.**TaskManager**(*context*, *node_id*, *shared=False*, *purpose='unspecified action'*, *retry=True*, *patient=False*, *load_driver=True*)

Bases: `object`

Context manager for tasks.

This class wraps the locking, driver loading, and acquisition of related resources (eg, Node and Ports) when beginning a unit of work.

downgrade_lock()
Downgrade the lock to a shared one.

load_driver()

property node

property portgroups

property ports

process_event(*event*, *callback=None*, *call_args=None*, *call_kwargs=None*, *err_handler=None*, *target_state=None*, *last_error=None*)
Process the given event for the tasks current state.

Parameters

- **event** the name of the event to process
- **callback** optional callback to invoke upon event transition
- **call_args** optional args to pass to the callback method
- **call_kwargs** optional kwargs to pass to the callback method
- **err_handler** optional error handler to invoke if the callback fails, eg. because there are no workers available (err_handler should accept arguments node, prev_prov_state, and prev_target_state)
- **target_state** if specified, the target provision state for the node. Otherwise, use the target state from the fsm
- **last_error** last error to set on the node together with the state transition.

Raises InvalidState if the event is not allowed by the associated state machine

release_resources()
Unlock a node and release resources.

If an exclusive lock is held, unlock the node. Reset attributes to make it clear that this instance of TaskManager should no longer be accessed.

resume_cleaning()
A helper to resume cleaning with the right target state.

set_spawn_error_hook(*_on_error_method*, **args*, ***kwargs*)
Create a hook to handle exceptions when spawning a task.

Create a hook that gets called upon an exception being raised from spawning a background thread to do a task.

Parameters

- **_on_error_method** a callable object, its first parameter should accept the Exception object that was raised.
- **args** additional args passed to the callable object.
- **kwargs** additional kwargs passed to the callable object.

spawn_after(*_spawn_method*, **args*, ***kwargs*)
Call this to spawn a thread to complete the task.

The specified method will be called when the TaskManager instance exits.

Parameters

- **_spawn_method** a method that returns a GreenThread object
- **args** args passed to the method.
- **kwargs** additional kwargs passed to the method.

upgrade_lock(*purpose=None*, *retry=None*)
Upgrade a shared lock to an exclusive lock.

Also reloads node object from the database. If lock is already exclusive only changes the lock purpose when provided with one.

Parameters

- **purpose** optionally change the purpose of the lock
- **retry** whether to retry locking if it fails, the class-level value is used by default

Raises NodeLocked if an exclusive lock remains on the node after node_locked_retry_attempts

property volume_connectors

property volume_targets

ironic.conductor.task_manager.**acquire**(*context*, **args*, ***kwargs*)
Shortcut for acquiring a lock on a Node.

Parameters **context** Request context.

Returns An instance of *TaskManager*.

ironic.conductor.task_manager.**require_exclusive_lock**(*f*)
Decorator to require an exclusive lock.

Decorated functions must take a *TaskManager* as the first parameter. Decorated class methods should take a *TaskManager* as the first parameter after self.

ironic.conductor.utils module

ironic.conductor.utils.**abort_on_conductor_take_over**(*task*)
Set nodes state when a task was aborted due to conductor take over.

Parameters **task** a TaskManager instance.

ironic.conductor.utils.**add_secret_token**(*node*, *pregenerated=False*)
Adds a secret token to driver_internal_info for IPA verification.

Parameters

- **node** Node object
- **pregenerated** Boolean value, default False, which indicates if the token should be marked as pregenerated in order to facilitate virtual media booting where the token is embedded into the configuration.

ironic.conductor.utils.**agent_is_alive**(*node*, *timeout=None*)
Check that the agent is likely alive.

The method then checks for the last agent heartbeat, and if it occured within the timeout set by [deploy]fast_track_timeout, then agent is presumed alive.

Parameters

- **node** A node object.
- **timeout** Heartbeat timeout, defaults to *fast_track_timeout*.

ironic.conductor.utils.**build_configdrive**(*node*, *configdrive*)

Build a configdrive from provided meta_data, network_data and user_data.

If uuid or name are not provided in the meta_data, theyre defauled to the nodes uuid and name accordingly.

Parameters

- **node** an Ironic node object.
- **configdrive** A configdrive as a dict with keys `meta_data`, `network_data`, `user_data` and `vendor_data` (all optional).

Returns A gzipped and base64 encoded configdrive as a string.

ironic.conductor.utils.**cleaning_error_handler**(*task*, *logmsg*, *errmsg=None*, *traceback=False*, *tear_down_cleaning=True*, *set_fail_state=True*, *set_maintenance=None*)

Put a failed node in CLEANFAIL and maintenance (if needed).

Parameters

- **task** a TaskManager instance.
- **logmsg** Message to be logged.
- **errmsg** Message for the user. Optional, if not provided *logmsg* is used.
- **traceback** Whether to log a traceback. Defaults to False.
- **tear_down_cleaning** Whether to clean up the PXE and DHCP files after cleaning. Default to True.
- **set_fail_state** Whether to set node to failed state. Default to True.
- **set_maintenance** Whether to set maintenance mode. If None, maintenance mode will be set if and only if a clean step is being executed on a node.

ironic.conductor.utils.**cleanup_after_timeout**(*task*)

Cleanup deploy task after timeout.

Parameters **task** a TaskManager instance.

ironic.conductor.utils.**cleanup_cleanwait_timeout**(*task*)

Cleanup a cleaning task after timeout.

Parameters **task** a TaskManager instance.

ironic.conductor.utils.**cleanup_rescuewait_timeout**(*task*)

Cleanup rescue task after timeout.

Parameters **task** a TaskManager instance.

ironic.conductor.utils.**deploying_error_handler**(*task*, *logmsg*, *errmsg=None*, *traceback=False*, *clean_up=True*)

Put a failed node in DEPLOYFAIL.

Parameters

- **task** the task
- **logmsg** message to be logged
- **errmsg** message for the user
- **traceback** Boolean; True to log a traceback
- **clean_up** Boolean; True to clean up

ironic.conductor.utils.**exclude_current_conductor**(*current_conductor*, *offline_conductors*)

Wrapper to exclude current conductor from offline_conductors

In some cases the current conductor may have failed to update the heartbeat timestamp due to failure or resource starvation. When this occurs the dbapi get_offline_conductors method will include the current conductor in its return value.

Parameters

- **current_conductor** id or hostname of the current conductor
- **offline_conductors** List of offline conductors.

Returns List of offline conductors, excluding current conductor

ironic.conductor.utils.**fail_on_error**(*error_callback*, *msg*, **error_args*, ***error_kwargs*)

A decorator for failing operation on failure.

ironic.conductor.utils.**fast_track_able**(*task*)

Checks if the operation can be a streamlined deployment sequence.

This is mainly focused on ensuring that we are able to quickly sequence through operations if we already have a ramdisk heartbeating through external means.

Parameters **task** Taskmanager object

Returns True if [deploy]fast_track is set to True, no iSCSI boot configuration is present, and no last_error is present for the node indicating that there was a recent failure.

ironic.conductor.utils.**get_attached_vif**(*port*)

Get any attached vif ID for the port

Parameters **port** The port object upon which to check for a vif record.

Returns Returns a tuple of the vif if found and the use of the vif in the form of a string, tenant, cleaning provisioning, rescuing.

Raises InvalidState exception upon finding a port with a transient state vif on the port.

ironic.conductor.utils.**get_configdrive_image**(*node*)

Get configdrive as an ISO image or a URL.

Converts the JSON representation into an image. URLs and raw contents are returned unchanged.

Parameters **node** an Ironic node object.

Returns A gzipped and base64 encoded configdrive as a string.

ironic.conductor.utils.**get_node_next_clean_steps**(*task*, *skip_current_step=True*)

ironic.conductor.utils.**get_node_next_deploy_steps**(*task*, *skip_current_step=True*)

ironic.conductor.utils.**hash_password**(*password=''*)
Hashes a supplied password.

Parameters **password** password to be hashed

ironic.conductor.utils.**is_agent_token_pregenerated**(*node*)
Determines if the token was generated for out of band configuration.

Ironic supports the ability to provide configuration data to the agent through the a virtual floppy or as part of the virtual media image which is attached to the BMC.

This method helps us identify WHEN we did so as we dont need to remove records of the token prior to rebooting the token. This is important as tokens provided through out of band means presist in the virtual media image, are loaded as part of the agent ramdisk, and do not require regeneration of the token upon the initial lookup, ultimately making the overall usage of virtual media and pregenerated tokens far more secure.

Parameters **node** Node Object

Returns True if the token was pregenerated as indicated by the nodes driver_internal_info field. False in all other cases.

ironic.conductor.utils.**is_agent_token_present**(*node*)
Determines if an agent token is present upon a node.

Parameters **node** Node object

Returns True if an agent_secret_token value is present in a node driver_internal_info field.

ironic.conductor.utils.**is_agent_token_valid**(*node*, *token*)
Validates if a supplied token is valid for the node.

Parameters

- **node** Node object
- **token** A token value to validate against the driver_internal_info field agent_secret_token.

Returns True if the supplied token matches the token recorded in the supplied node object.

ironic.conductor.utils.**is_fast_track**(*task*)
Checks a fast track is available.

This method first ensures that the node and conductor configuration is valid to perform a fast track sequence meaning that we already have a ramdisk running through another means like discovery. If not valid, False is returned.

The method then checks for the last agent heartbeat, and if it occured within the timeout set by [deploy]fast_track_timeout and the power state for the machine is POWER_ON, then fast track is permitted.

Parameters **task** Taskmanager object

Returns True if the last heartbeat that was recorded was within the [deploy]fast_track_timeout setting.

ironic.conductor.utils.**make_salt**()
Generate a random salt with the indicator tag for password type.

Returns a valid salt for use with crypt.crypt

ironic.conductor.utils.**node_cache_bios_settings**(*task*, *node*)
Do caching of bios settings if supported by driver

ironic.conductor.utils.**node_cache_boot_mode**(*task*)
Cache boot_mode and secure_boot state if supported by driver.

Cache current boot_mode and secure_boot in ironics node representation

Parameters **task** a TaskManager instance containing the node to check.

ironic.conductor.utils.**node_cache_vendor**(*task*)
Cache the vendor if it can be detected.

ironic.conductor.utils.**node_change_boot_mode**(*task*, *target_boot_mode*)
Change boot mode to requested state for node

Parameters

- **task** a TaskManager instance containing the node to act on.
- **target_boot_mode** Any boot mode in *ironic.common.boot_modes*.

ironic.conductor.utils.**node_change_secure_boot**(*task*, *secure_boot_target*)
Change secure_boot state to requested state for node

Parameters

- **task** a TaskManager instance containing the node to act on.
- **secure_boot_target** (*boolean*) Target secure_boot state OneOf(True => on, False => off)

ironic.conductor.utils.**node_get_boot_mode**(*task*)
Read currently set boot mode from a node.

Reads the boot mode for a node. If boot mode cant be discovered, *None* is returned.

Parameters **task** a TaskManager instance.

Raises DriverOperationError or its derivative in case of driver runtime error.

Raises UnsupportedDriverExtension if current driver does not have management interface or *get_boot_mode()* method is not supported.

Returns Boot mode. One of `ironic.common.boot_mode` or *None* if boot mode cant be discovered

ironic.conductor.utils.**node_history_record**(*node*, *conductor=None*, *event=None*, *event_type=None*, *user=None*, *error=False*)
Records a node history record

Adds an entry to the node history table with the appropriate fields populated to ensure consistent experience by also updating the node `last_error` field. Please note the event is only recorded if the `[conductor]node_history_max_size` parameter is set to a value greater than `0`.

Parameters

- **node** A node object from a task object. Required.
- **conductor** The hostname of the conductor. If not specified this value is populated with the conductor FQDN.
- **event** The text to record to the node history table. If no value is supplied, the method silently returns to the caller.
- **event_type** The type activity where the event was encountered, either provisioning, monitoring, cleaning, or whatever text the a driver author wishes to supply based upon the activity. The purpose is to help guide an API consumer/operator to have a better contextual understanding of what was going on *when* the event occured.
- **user** The user_id value which triggered the request, if available.
- **error** Boolean value, default false, to signify if the event is an error which should be recorded in the node `last_error` field.

Returns None. No value is returned by this method.

ironic.conductor.utils.**node_power_action**(*task*, *new_state*, *timeout=None*)

Change power state or reset for a node.

Perform the requested power action if the transition is required.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **new_state** Any power state from ironic.common.states.
- **timeout** timeout (in seconds) positive integer (> 0) for any power state. `None` indicates to use default timeout.

Raises InvalidParameterValue when the wrong state is specified or the wrong driver info is specified.

Raises StorageError when a failure occurs updating the nodes storage interface upon setting power on.

Raises other exceptions by the nodes power driver if something wrong occurred during the power action.

ironic.conductor.utils.**node_set_boot_device**(*task*, *device*, *persistent=False*)

Set the boot device for a node.

If the node that the boot device change is being requested for is in ADOPTING state, the boot device will not be set as that change could potentially result in the future running state of an adopted node being modified erroneously.

Parameters

- **task** a TaskManager instance.
- **device** Boot device. Values are vendor-specific.
- **persistent** Whether to set next-boot, or make the change permanent. Default: False.

Raises InvalidParameterValue if the validation of the ManagementInterface fails.

ironic.conductor.utils.**node_set_boot_mode**(*task*, *mode*)
Set the boot mode for a node.

Sets the boot mode for a node if the nodes driver interface contains a management interface.

If the node that the boot mode change is being requested for is in ADOPTING state, the boot mode will not be set as that change could potentially result in the future running state of an adopted node being modified erroneously.

Parameters

- **task** a TaskManager instance.
- **mode** Boot mode. Values are one of *ironic.common.boot_modes*

Raises InvalidParameterValue if the validation of the ManagementInterface fails.

Raises DriverOperationError or its derivative in case of driver runtime error.

Raises UnsupportedDriverExtension if current driver does not have vendor interface or method is unsupported.

ironic.conductor.utils.**node_wait_for_power_state**(*task*, *new_state*, *timeout=None*)
Wait for node to be in new power state.

Parameters

- **task** a TaskManager instance.
- **new_state** the desired new power state, one of the power states in *ironic.common.states*.
- **timeout** number of seconds to wait before giving up. If not specified, uses the conductor.power_state_change_timeout config value.

Raises PowerStateFailure if timed out

ironic.conductor.utils.**notify_conductor_resume_clean**(*task*)

ironic.conductor.utils.**notify_conductor_resume_deploy**(*task*)

ironic.conductor.utils.**notify_conductor_resume_operation**(*task*, *operation*)
Notify the conductor to resume an operation.

Parameters

- **task** the task
- **operation** the operation, a string

ironic.conductor.utils.**power_on_node_if_needed**(*task*)
Powers on node if it is powered off and has a Smart NIC port

Parameters **task** A TaskManager object

Returns the previous power state or None if no changes were made

Raises exception.NetworkError if agent status didnt match the required status after max retry attempts.

ironic.conductor.utils.**power_state_error_handler**(*e*, *node*, *power_state*)
Set the nodes power states if error occurs.

This hook gets called upon an exception being raised when spawning the worker thread to change the power state of a node.

Parameters

- **e** the exception object that was raised.
- **node** an Ironic node object.
- **power_state** the power state to set on the node.

ironic.conductor.utils.**power_state_for_network_configuration**(*task*)
Handle the power state for a node reconfiguration.

Powers the node on if and only if it has a Smart NIC port. Yields for the actual reconfiguration, then restores the power state.

Parameters **task** A TaskManager object.

ironic.conductor.utils.**provisioning_error_handler**(*e*, *node*, *provision_state*, *target_provision_state*)
Set the nodes provisioning states if error occurs.

This hook gets called upon an exception being raised when spawning the worker to do some provisioning to a node like deployment, tear down, or cleaning.

Parameters

- **e** the exception object that was raised.
- **node** an Ironic node object.
- **provision_state** the provision state to be set on the node.
- **target_provision_state** the target provision state to be set on the node.

ironic.conductor.utils.**remove_agent_url**(*node*)
Helper to remove the agent_url record.

ironic.conductor.utils.**remove_node_rescue_password**(*node*, *save=True*)
Helper to remove rescue password from a node.

Removes rescue password from node. It saves node by default. If node should not be saved, then caller needs to explicitly indicate it.

Parameters

- **node** an Ironic node object.
- **save** Boolean; True (default) to save the node; False otherwise.

ironic.conductor.utils.**rescuing_error_handler**(*task*, *msg*, *set_fail_state=True*)
Cleanup rescue task after timeout or failure.

Parameters

- **task** a TaskManager instance.
- **msg** a message to set into nodes last_error field
- **set_fail_state** a boolean flag to indicate if node needs to be transitioned to a failed state. By default node would be transitioned to a failed state.

ironic.conductor.utils.**restore_power_state_if_needed**(*task*, *power_state_to_restore*)
Change the nodes power state if power_state_to_restore is not None

Parameters

- **task** A TaskManager object
- **power_state_to_restore** power state

ironic.conductor.utils.**skip_automated_cleaning**(*node*, *log=True*)
Checks if node cleaning needs to be skipped for an specific node.

Parameters **node** the node to consider

ironic.conductor.utils.**spawn_cleaning_error_handler**(*e*, *node*)
Handle spawning error for node cleaning.

ironic.conductor.utils.**spawn_deploying_error_handler**(*e*, *node*)
Handle spawning error for node deploying.

ironic.conductor.utils.**spawn_rescue_error_handler**(*e*, *node*)
Handle spawning error for node rescue.

ironic.conductor.utils.**store_agent_certificate**(*node*, *agent_verify_ca*)
Store certificate received from the agent and return its path.

ironic.conductor.utils.**update_image_type**(*context*, *node*)
Updates is_whole_disk_image and image_type based on the node data.

Parameters

- **context** Request context.
- **node** Node object.

Returns True if any changes have been done, else False.

ironic.conductor.utils.**update_next_step_index**(*task*, *step_type*)
Calculate the next step index and update the node.

Parameters

- **task** A TaskManager object
- **step_type** The type of steps to process: clean or deploy.

Returns Index of the next step.

ironic.conductor.utils.**validate_instance_info_traits**(*node*)
Validate traits in instance_info.

All traits in instance_info must also exist as node traits.

Parameters **node** an Ironic node object.

Raises InvalidParameterValue if the instance traits are badly formatted, or contain traits that are not set on the node.

ironic.conductor.utils.**validate_port_physnet**(*task*, *port_obj*)
Validate the consistency of physical networks of ports in a portgroup.

Validate the consistency of a ports physical network with other ports in the same portgroup. All ports in a portgroup should have the same value (which may be None) for their physical_network field.

During creation or update of a port in a portgroup we apply the following validation criteria:

- If the portgroup has existing ports with different physical networks, we raise PortgroupPhysnetInconsistent. This shouldnt ever happen.
- If the port has a physical network that is inconsistent with other ports in the portgroup, we raise exception.Conflict.

If a ports physical network is None, this indicates that ironics VIF attachment mapping algorithm should operate in a legacy (physical network unaware) mode for this port or portgroup. This allows existing ironic nodes to continue to function after an upgrade to a release including physical network support.

Parameters

- **task** a TaskManager instance
- **port_obj** a port object to be validated.

Raises Conflict if the port is a member of a portgroup which is on a different physical network.

Raises PortgroupPhysnetInconsistent if the ports portgroup has ports which are not all assigned the same physical network.

ironic.conductor.utils.**value_within_timeout**(*value*, *timeout*)

Checks if the time is within the previous timeout seconds from now.

Parameters

- **value** a string representing date and time or None.
- **timeout** timeout in seconds.

ironic.conductor.utils.**verifying_error_handler**(*task*, *logmsg*, *errmsg=None*, *traceback=False*)

Handle errors during verification steps

Parameters

- **task** the task
- **logmsg** message to be logged
- **errmsg** message for the user
- **traceback** Boolean; True to log a traceback

ironic.conductor.utils.**wipe_cleaning_internal_info**(*task*)

Remove temporary cleaning fields from driver_internal_info.

ironic.conductor.utils.**wipe_deploy_internal_info**(*task*)

Remove temporary deployment fields from driver_internal_info.

ironic.conductor.utils.**wipe_internal_info_on_power_off**(*node*)

Wipe information that should not survive reboot/power off.

ironic.conductor.utils.**wipe_token_and_url**(*task*)

Remove agent URL and token from the task.

ironic.conductor.verify module

Functionality related to verify steps.

ironic.conductor.verify.**do_node_verify**(*task*)
Internal method to perform power credentials verification.

Module contents

ironic.conf package

Submodules

ironic.conf.agent module

ironic.conf.agent.**register_opts**(*conf*)

ironic.conf.anaconda module

ironic.conf.anaconda.**register_opts**(*conf*)

ironic.conf.ansible module

ironic.conf.ansible.**register_opts**(*conf*)

ironic.conf.api module

class ironic.conf.api.**Octal**(*min=None*, *max=None*, *type_name='integer value'*, *choices=None*)
Bases: oslo_config.types.Integer

ironic.conf.api.**register_opts**(*conf*)

ironic.conf.audit module

ironic.conf.audit.**register_opts**(*conf*)

ironic.conf.auth module

ironic.conf.auth.**add_auth_opts**(*options*, *service_type=None*)
Add auth options to sample config

As these are dynamically registered at runtime, this adds options for most used auth_plugins when generating sample config.

ironic.conf.auth.**register_auth_opts**(*conf*, *group*, *service_type=None*)
Register session- and auth-related options

Registers only basic auth options shared by all auth plugins. The rest are registered at runtime depending on auth plugin used.

ironic.conf.cinder module

ironic.conf.cinder.**list_opts**()

ironic.conf.cinder.**register_opts**(*conf*)

ironic.conf.conductor module

ironic.conf.conductor.**register_opts**(*conf*)

ironic.conf.console module

ironic.conf.console.**register_opts**(*conf*)

ironic.conf.database module

ironic.conf.database.**register_opts**(*conf*)

ironic.conf.default module

ironic.conf.default.**list_opts**()

ironic.conf.default.**register_opts**(*conf*)

ironic.conf.deploy module

ironic.conf.deploy.**register_opts**(*conf*)

ironic.conf.dhcp module

ironic.conf.dhcp.**register_opts**(*conf*)

ironic.conf.drac module

ironic.conf.drac.**register_opts**(*conf*)

ironic.conf.glance module

ironic.conf.glance.**list_opts**()

ironic.conf.glance.**register_opts**(*conf*)

ironic.conf.healthcheck module

ironic.conf.healthcheck.**register_opts**(*conf*)

ironic.conf.ibmc module

ironic.conf.ibmc.**register_opts**(*conf*)

ironic.conf.ilo module

ironic.conf.ilo.**register_opts**(*conf*)

ironic.conf.inspector module

ironic.conf.inspector.**list_opts**()

ironic.conf.inspector.**register_opts**(*conf*)

ironic.conf.ipmi module

ironic.conf.ipmi.**register_opts**(*conf*)

ironic.conf.irmc module

ironic.conf.irmc.**register_opts**(*conf*)

ironic.conf.metrics module

ironic.conf.metrics.**register_opts**(*conf*)

ironic.conf.metrics_statsd module

ironic.conf.metrics_statsd.**register_opts**(*conf*)

ironic.conf.molds module

ironic.conf.molds.**register_opts**(*conf*)

ironic.conf.neutron module

ironic.conf.neutron.**list_opts**()

ironic.conf.neutron.**register_opts**(*conf*)

ironic.conf.nova module

ironic.conf.nova.**list_opts**()

ironic.conf.nova.**register_opts**(*conf*)

ironic.conf.opts module

ironic.conf.opts.**list_opts**()

Return a list of oslo.config options available in Ironic code.

The returned list includes all oslo.config options. Each element of the list is a tuple. The first element is the name of the group, the second element is the options.

The function is discoverable via the ironic entry point under the oslo.config.opts namespace.

The function is used by Oslo sample config file generator to discover the options.

Returns a list of (group, options) tuples

ironic.conf.opts.**update_opt_defaults**()

ironic.conf.pxe module

ironic.conf.pxe.**register_opts**(*conf*)

ironic.conf.redfish module

ironic.conf.redfish.**register_opts**(*conf*)

ironic.conf.service_catalog module

ironic.conf.service_catalog.**list_opts**()

ironic.conf.service_catalog.**register_opts**(*conf*)

ironic.conf.snmp module

ironic.conf.snmp.**register_opts**(*conf*)

ironic.conf.swift module

ironic.conf.swift.**list_opts**()

ironic.conf.swift.**register_opts**(*conf*)

ironic.conf.xclarity module

ironic.conf.xclarity.**register_opts**(*conf*)

Module contents

ironic.db package

Subpackages

ironic.db.sqlalchemy package

Submodules

ironic.db.sqlalchemy.api module

SQLAlchemy storage backend.

class ironic.db.sqlalchemy.api.**Connection**

Bases: *ironic.db.api.Connection*

SqlAlchemy connection.

add_node_tag(*node_id*, *tag*)
Add tag to the node.

If the node_id and tag pair already exists, this should still succeed.

Parameters

- **node_id** The id of a node.
- **tag** A tag string.

Returns the NodeTag object.

Raises NodeNotFound if the node is not found.

add_node_trait(*node_id*, *trait*, *version*)
Add trait to the node.

If the node_id and trait pair already exists, this should still succeed.

Parameters

- **node_id** The id of a node.
- **trait** A trait string.
- **version** the version of the object.Trait.

Returns the NodeTrait object.

Raises InvalidParameterValue if adding the trait would exceed the per-node traits limit.

Raises NodeNotFound if the node is not found.

bulk_delete_node_history_records(*entries*)
Utility method to bulk delete node history entries.

Parameters **entires** A list of node history entriy ids to be queried for deletion.

check_node_list(*idents*, *project=None*)
Check a list of node identities and map it to UUIDs.

This call takes a list of node names and/or UUIDs and tries to convert them to UUIDs. It fails early if any identities cannot possible be used as names or UUIDs.

Parameters **idents** List of identities.

Returns A mapping from requests identities to node UUIDs.

Raises NodeNotFound if some identities were not found or cannot be valid names or UUIDs.

check_versions(*ignore_models=()*, *permit_initial_version=False*)
Checks the whole database for incompatible objects.

This scans all the tables in search of objects that are not supported; i.e., those that are not specified in *ironic.common.release_mappings.RELEASE_MAPPING*. This includes objects that have null version values.

Parameters

- **ignore_models** List of model names to skip.

- **permit_initial_version** Boolean, default False, to permit a NoSuchTableError exception to be raised by SQLAlchemy and accordingly bypass when an object has its initial object version.

Returns A Boolean. True if all the objects have supported versions; False otherwise.

clear_node_reservations_for_conductor(*hostname*)

clear_node_target_power_state(*hostname*)

create_allocation(*values*)
Create a new allocation.

Parameters **values** Dict of values to create an allocation with

Returns An allocation

Raises AllocationDuplicateName

Raises AllocationAlreadyExists

create_bios_setting_list(*node_id*, *settings*, *version*)
Create a list of BIOSSetting records for a given node.

Parameters

- **node_id** The node id.
- **settings** A list of BIOS Settings to be created.

```
[
  {
   'name': String,
   'value': String,
   additional settings from BIOS registry
  },
  {
   'name': String,
   'value': String,
   additional settings from BIOS registry
  },
  ...
]
```

- **version** the version of the object.BIOSSetting.

Returns A list of BIOSSetting object.

Raises NodeNotFound if the node is not found.

Raises BIOSSettingAlreadyExists if any of the setting records already exists.

create_chassis(*values*)
Create a new chassis.

Parameters **values** Dict of values.

create_deploy_template(*values*)
Create a deployment template.

Parameters **values** A dict describing the deployment template. For example:

```
{
 'uuid': uuidutils.generate_uuid(),
 'name': 'CUSTOM_DT1',
}
```

Raises DeployTemplateDuplicateName if a deploy template with the same name exists.

Raises DeployTemplateAlreadyExists if a deploy template with the same UUID exists.

Returns A deploy template.

create_node(*values*)
Create a new node.

Parameters **values** A dict containing several items used to identify and track the node, and several dicts which are passed into the Drivers when managing this node. For example:

```
{
 'uuid': uuidutils.generate_uuid(),
 'instance_uuid': None,
 'power_state': states.POWER_OFF,
 'provision_state': states.AVAILABLE,
 'driver': 'ipmi',
 'driver_info': { ... },
 'properties': { ... },
 'extra': { ... },
}
```

Raises InvalidParameterValue if values contains tags or traits.

Returns A node.

create_node_history(*values*)
Create a new history record.

Parameters **values** Dict of values.

create_port(*values*)
Create a new port.

Parameters **values** Dict of values.

create_portgroup(*values*)
Create a new portgroup.

Parameters **values** Dict of values with the following keys: id uuid name node_id address extra created_at updated_at

Returns A portgroup

Raises PortgroupDuplicateName

Raises PortgroupMACAlreadyExists

Raises PortgroupAlreadyExists

create_volume_connector(*connector_info*)
Create a new volume connector.

Parameters **connector_info** Dictionary containing information about the connector. Example:

```
{
    'uuid': '000000-..',
    'type': 'wwnn',
    'connector_id': '00:01:02:03:04:05:06',
    'node_id': 2
}
```

Returns A volume connector.

Raises VolumeConnectorTypeAndIdAlreadyExists If a connector already exists with a matching type and connector_id.

Raises VolumeConnectorAlreadyExists If a volume connector with the same UUID already exists.

create_volume_target(*target_info*)
Create a new volume target.

Parameters **target_info** Dictionary containing the information about the volume target. Example:

```
{
    'uuid': '000000-..',
    'node_id': 2,
    'boot_index': 0,
    'volume_id': '12345678-...'
    'volume_type': 'some type',
}
```

Returns A volume target.

Raises VolumeTargetBootIndexAlreadyExists if a volume target already exists with the same boot index and node ID.

Raises VolumeTargetAlreadyExists if a volume target with the same UUID exists.

delete_bios_setting_list(*node_id*, *names*)
Delete a list of BIOS settings.

Parameters

- **node_id** The node id.
- **names** List of BIOS setting names to be deleted.

Raises NodeNotFound if the node is not found.

Raises BIOSSettingNotFound if any of BIOS setting name is not found.

delete_node_tag(*node_id*, *tag*)
Delete specified tag from the node.

Parameters

- **node_id** The id of a node.
- **tag** A tag string.

Raises NodeNotFound if the node is not found.

Raises NodeTagNotFound if the tag is not found.

delete_node_trait(*node_id*, *trait*)
Delete specified trait from the node.

Parameters

- **node_id** The id of a node.
- **trait** A trait string.

Raises NodeNotFound if the node is not found.

Raises NodeTraitNotFound if the trait is not found.

destroy_allocation(*allocation_id*)
Destroy an allocation.

Parameters **allocation_id** Allocation ID or UUID

Raises AllocationNotFound

destroy_chassis(*chassis_id*)
Destroy a chassis.

Parameters **chassis_id** The id or the uuid of a chassis.

destroy_deploy_template(*template_id*)
Destroy a deployment template.

Parameters **template_id** ID of the deployment template to destroy.

Raises DeployTemplateNotFound if the deploy template does not exist.

destroy_node(*node_id*)
Destroy a node and its associated resources.

Destroy a node, including any associated ports, port groups, tags, traits, volume connectors, and volume targets.

Parameters **node_id** The ID or UUID of a node.

destroy_node_history_by_uuid(*history_uuid*)
Destroy a history record.

Parameters **history_uuid** The uuid of a history record

destroy_port(*port_id*)
Destroy an port.

Parameters **port_id** The id or MAC of a port.

destroy_portgroup(*portgroup_id*)
Destroy a portgroup.

Parameters **portgroup_id** The UUID or MAC of a portgroup.

Raises PortgroupNotEmpty

Raises PortgroupNotFound

destroy_volume_connector(*ident*)
Destroy a volume connector.

Parameters **ident** The UUID or integer ID of a volume connector.

Raises VolumeConnectorNotFound If a volume connector with the specified ident does not exist.

destroy_volume_target(*ident*)
Destroy a volume target.

Parameters **ident** The UUID or integer ID of a volume target.

Raises VolumeTargetNotFound if a volume target with the specified ident does not exist.

get_active_hardware_type_dict(*use_groups=False*)
Retrieve hardware types for the registered and active conductors.

Parameters **use_groups** Whether to factor conductor_group into the keys.

Returns

A dict which maps hardware type names to the set of hosts which support them. For example:

```
{hardware-type-a: set([host1, host2]),
 hardware-type-b: set([host2, host3])}
```

get_allocation_by_id(*allocation_id*)
Return an allocation representation.

Parameters **allocation_id** The id of an allocation.

Returns An allocation.

Raises AllocationNotFound

get_allocation_by_name(*name*)
Return an allocation representation.

Parameters **name** The logical name of an allocation.

Returns An allocation.

Raises AllocationNotFound

get_allocation_by_uuid(*allocation_uuid*)
Return an allocation representation.

Parameters **allocation_uuid** The uuid of an allocation.

Returns An allocation.

Raises AllocationNotFound

get_allocation_list(*filters=None*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)
Return a list of allocations.

Parameters

- **filters** Filters to apply. Defaults to None.

  node_uuid uuid of node

  state allocation state

  resource_class requested resource class

- **limit** Maximum number of allocations to return.
- **marker** The last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** Direction in which results should be sorted. (asc, desc)

Returns A list of allocations.

get_bios_setting(*node_id*, *name*)
Retrieve BIOS setting value.

Parameters

- **node_id** The node id.
- **name** String containing name of BIOS setting to be retrieved.

Returns The BIOSSetting object.

Raises NodeNotFound if the node is not found.

Raises BIOSSettingNotFound if the BIOS setting is not found.

get_bios_setting_list(*node_id*)
Retrieve BIOS settings of a given node.

Parameters **node_id** The node id.

Returns A list of BIOSSetting objects.

Raises NodeNotFound if the node is not found.

get_chassis_by_id(*chassis_id*)
Return a chassis representation.

Parameters **chassis_id** The id of a chassis.

Returns A chassis.

get_chassis_by_uuid(*chassis_uuid*)
Return a chassis representation.

Parameters **chassis_uuid** The uuid of a chassis.

Returns A chassis.

get_chassis_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)
Return a list of chassis.

Parameters

- **limit** Maximum number of chassis to return.
- **marker** the last item of the previous page; we return the next result set.

- **sort_key** Attribute by which results should be sorted.
- **sort_dir** direction in which results should be sorted. (asc, desc)

get_conductor(*hostname*, *online=True*)
Retrieve a conductors service record from the database.

Parameters

- **hostname** The hostname of the conductor service.
- **online** Specify the filter value on the *online* field when querying conductors. The online field is ignored if this value is set to None.

Returns A conductor.

Raises ConductorNotFound if the conductor with given hostname does not exist or doesnt meet the specified online expectation.

get_conductor_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)
Return a list of conductors.

Parameters

- **limit** Maximum number of conductors to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** direction in which results should be sorted. (asc, desc)

get_deploy_template_by_id(*template_id*)
Retrieve a deployment template by ID.

Parameters **template_id** ID of the deployment template to retrieve.

Raises DeployTemplateNotFound if the deploy template does not exist.

Returns A deploy template.

get_deploy_template_by_name(*template_name*)
Retrieve a deployment template by name.

Parameters **template_name** name of the deployment template to retrieve.

Raises DeployTemplateNotFound if the deploy template does not exist.

Returns A deploy template.

get_deploy_template_by_uuid(*template_uuid*)
Retrieve a deployment template by UUID.

Parameters **template_uuid** UUID of the deployment template to retrieve.

Raises DeployTemplateNotFound if the deploy template does not exist.

Returns A deploy template.

get_deploy_template_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)
Retrieve a list of deployment templates.

Parameters

- **limit** Maximum number of deploy templates to return.

- **marker** The last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** Direction in which results should be sorted. (asc, desc)

Returns A list of deploy templates.

get_deploy_template_list_by_names(*names*)
Return a list of deployment templates with one of a list of names.

Parameters **names** List of names to filter by.

Returns A list of deploy templates.

get_node_by_id(*node_id*)
Return a node.

Parameters **node_id** The id of a node.

Returns A node.

get_node_by_instance(*instance*)
Return a node.

Parameters **instance** The instance uuid to search for.

Returns A node.

Raises InstanceNotFound if the instance is not found.

Raises InvalidUUID if the instance uuid is invalid.

get_node_by_name(*node_name*)
Return a node.

Parameters **node_name** The logical name of a node.

Returns A node.

get_node_by_port_addresses(*addresses*)
Find a node by any matching port address.

Parameters **addresses** list of port addresses (e.g. MACs).

Returns Node object.

Raises NodeNotFound if none or several nodes are found.

get_node_by_uuid(*node_uuid*)
Return a node.

Parameters **node_uuid** The uuid of a node.

Returns A node.

get_node_history_by_id(*history_id*)
Return a node history representation.

Parameters **history_id** The id of a history record.

Returns A history.

get_node_history_by_node_id(*node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)

List all the history records for a given node.

Parameters

- **node_id** The integer node ID.
- **limit** Maximum number of history records to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted
- **sort_dir** direction in which results should be sorted (asc, desc)

Returns A list of histories.

get_node_history_by_uuid(*history_uuid*)

Return a node history representation.

Parameters **history_uuid** The uuid of a history record

Returns A history.

get_node_history_list(*limit=None*, *marker=None*, *sort_key='created_at'*, *sort_dir='asc'*)

Return a list of node history records

Parameters

- **limit** Maximum number of history records to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** direction in which results should be sorted. (asc, desc)

get_node_list(*filters=None*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *fields=None*)

Return a list of nodes.

Parameters

- **filters** Filters to apply. Defaults to None.

  associated True | False

  reserved True | False

  maintenance True | False

  chassis_uuid uuid of chassis

  driver drivers name

  provision_state provision state of node

  provisioned_before nodes with provision_updated_at field before this interval in seconds

- **limit** Maximum number of nodes to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.

- **sort_dir** direction in which results should be sorted. (asc, desc)
- **fields** Comma separated field list to return, to allow for only specific fields to be returned to have maximum API performance calls where not all columns are needed from the database.

get_node_list_columns(*columns=None*, *filters=None*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)

Get a node list with specific fields/columns.

Parameters

- **columns** A list of columns to retrieve from the database and populate into the object.
- **filters** The requested database field filters in the form of a dictionary with the applicable key, and filter value.
- **limit** Limit the number of returned nodes, default None.
- **marker** Starting marker to generate a paginated result set for the consumer.
- **sort_key** Sort key to apply to the result set.
- **sort_dir** Sort direction to apply to the result set.

Returns A list of Node objects based on the data model from a SQLAlchemy result set, which the object layer can use to convert the node into an Node object list.

get_node_tags_by_node_id(*node_id*)

Get node tags based on its id.

Parameters **node_id** The id of a node.

Returns A list of NodeTag objects.

Raises NodeNotFound if the node is not found.

get_node_traits_by_node_id(*node_id*)

Get node traits based on its id.

Parameters **node_id** The id of a node.

Returns A list of NodeTrait objects.

Raises NodeNotFound if the node is not found.

get_nodeinfo_list(*columns=None*, *filters=None*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)

Get specific columns for matching nodes.

Return a list of the specified columns for all nodes that match the specified filters.

Parameters

- **columns** List of column names to return. Defaults to id column when columns == None.
- **filters** Filters to apply. Defaults to None.

  associated True | False

  chassis_uuid uuid of chassis

conductor_group conductor group name

console_enabled True | False

description_contains substring in description

driver drivers name

fault current fault type

id numeric ID

inspection_started_before nodes with inspection_started_at field before this interval in seconds

instance_uuid uuid of instance

lessee nodes lessee (e.g. project ID)

maintenance True | False

owner nodes owner (e.g. project ID)

project either owner or lessee

reserved True | False

reserved_by_any_of [conductor1, conductor2]

resource_class resource class name

retired True | False

provision_state provision state of node

provision_state_in provision state of node (multiple possibilities)

provisioned_before nodes with provision_updated_at field before this interval in seconds

uuid uuid of node

uuid_in uuid of node (multiple possibilities)

with_power_state True | False

- **limit** Maximum number of nodes to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** direction in which results should be sorted. (asc, desc)

Returns A list of tuples of the specified columns.

get_not_versions(*model_name*, *versions*)

Returns objects with versions that are not the specified versions.

This returns objects with versions that are not the specified versions. Objects with null versions (there shouldnt be any) are also returned.

Parameters

- **model_name** the name of the model (class) of desired objects
- **versions** list of versions of objects not to be returned

Returns list of the DB objects

Raises IronicException if there is no class associated with the name

get_offline_conductors(*field='hostname'*)
Get a list conductors that are offline (dead).

Parameters **field** A field to return, hostname by default.

Returns A list of requested fields of offline conductors.

get_online_conductors()
Get a list conductor hostnames that are online and active.

Returns A list of conductor hostnames.

get_port_by_address(*address*, *owner=None*, *project=None*)
Return a network port representation.

Parameters **address** The MAC address of a port.

Returns A port.

get_port_by_id(*port_id*)
Return a network port representation.

Parameters **port_id** The id of a port.

Returns A port.

get_port_by_name(*port_name*)
Return a network port representation.

Parameters **port_name** The name of a port.

Returns A port.

get_port_by_uuid(*port_uuid*)
Return a network port representation.

Parameters **port_uuid** The uuid of a port.

Returns A port.

get_port_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *owner=None*, *project=None*)
Return a list of ports.

Parameters

- **limit** Maximum number of ports to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** direction in which results should be sorted. (asc, desc)

get_portgroup_by_address(*address*, *project=None*)
Return a network portgroup representation.

Parameters

- **address** The MAC address of a portgroup.

- **project** A node owner or lessee to filter by.

Returns A portgroup.

Raises PortgroupNotFound

get_portgroup_by_id(*portgroup_id*, *project=None*)
Return a network portgroup representation.

Parameters **portgroup_id** The id of a portgroup.

Returns A portgroup.

Raises PortgroupNotFound

get_portgroup_by_name(*name*)
Return a network portgroup representation.

Parameters **name** The logical name of a portgroup.

Returns A portgroup.

Raises PortgroupNotFound

get_portgroup_by_uuid(*portgroup_uuid*)
Return a network portgroup representation.

Parameters **portgroup_uuid** The uuid of a portgroup.

Returns A portgroup.

Raises PortgroupNotFound

get_portgroup_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)
Return a list of portgroups.

Parameters

- **limit** Maximum number of portgroups to return.
- **marker** The last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** Direction in which results should be sorted. (asc, desc)
- **project** A node owner or lessee to filter by.

Returns A list of portgroups.

get_portgroups_by_node_id(*node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)
List all the portgroups for a given node.

Parameters

- **node_id** The integer node ID.
- **limit** Maximum number of portgroups to return.
- **marker** The last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted
- **sort_dir** Direction in which results should be sorted (asc, desc)

- **project** A node owner or lessee to filter by.

Returns A list of portgroups.

get_ports_by_node_id(*node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *owner=None*, *project=None*)

List all the ports for a given node.

Parameters

- **node_id** The integer node ID.
- **limit** Maximum number of ports to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted
- **sort_dir** direction in which results should be sorted (asc, desc)

Returns A list of ports.

get_ports_by_portgroup_id(*portgroup_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *owner=None*, *project=None*)

List all the ports for a given portgroup.

Parameters

- **portgroup_id** The integer portgroup ID.
- **limit** Maximum number of ports to return.
- **marker** The last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted
- **sort_dir** Direction in which results should be sorted (asc, desc)

Returns A list of ports.

get_volume_connector_by_id(*db_id*)

Return a volume connector representation.

Parameters **db_id** The integer database ID of a volume connector.

Returns A volume connector with the specified ID.

Raises VolumeConnectorNotFound If a volume connector with the specified ID is not found.

get_volume_connector_by_uuid(*connector_uuid*)

Return a volume connector representation.

Parameters **connector_uuid** The UUID of a connector.

Returns A volume connector with the specified UUID.

Raises VolumeConnectorNotFound If a volume connector with the specified UUID is not found.

get_volume_connector_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)

Return a list of volume connectors.

Parameters

- **limit** Maximum number of volume connectors to return.
- **marker** The last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** Direction in which results should be sorted. (asc, desc)
- **project** The associated node project to search with.

Returns a list of `VolumeConnector` objects

Returns A list of volume connectors.

Raises InvalidParameterValue If sort_key does not exist.

get_volume_connectors_by_node_id(*node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)

List all the volume connectors for a given node.

Parameters

- **node_id** The integer node ID.
- **limit** Maximum number of volume connectors to return.
- **marker** The last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted
- **sort_dir** Direction in which results should be sorted (asc, desc)
- **project** The associated node project to search with.

Returns a list of `VolumeConnector` objects

Returns A list of volume connectors.

Raises InvalidParameterValue If sort_key does not exist.

get_volume_target_by_id(*db_id*)

Return a volume target representation.

Parameters **db_id** The database primary key (integer) ID of a volume target.

Returns A volume target.

Raises VolumeTargetNotFound if no volume target with this ID exists.

get_volume_target_by_uuid(*uuid*)

Return a volume target representation.

Parameters **uuid** The UUID of a volume target.

Returns A volume target.

Raises VolumeTargetNotFound if no volume target with this UUID exists.

get_volume_target_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)

Return a list of volume targets.

Parameters

- **limit** Maximum number of volume targets to return.

- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** direction in which results should be sorted. (asc, desc)
- **project** The associated node project to search with.

Returns a list of `VolumeConnector` objects

Returns A list of volume targets.

Raises InvalidParameterValue if sort_key does not exist.

get_volume_targets_by_node_id(*node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)

List all the volume targets for a given node.

Parameters

- **node_id** The integer node ID.
- **limit** Maximum number of volume targets to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted
- **sort_dir** direction in which results should be sorted (asc, desc)
- **project** The associated node project to search with.

Returns a list of `VolumeConnector` objects

Returns A list of volume targets.

Raises InvalidParameterValue if sort_key does not exist.

get_volume_targets_by_volume_id(*volume_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)

List all the volume targets for a given volume id.

Parameters

- **volume_id** The UUID of the volume.
- **limit** Maximum number of volume targets to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted
- **sort_dir** direction in which results should be sorted (asc, desc)

Returns A list of volume targets.

Raises InvalidParameterValue if sort_key does not exist.

list_conductor_hardware_interfaces(*conductor_id*)

List all registered hardware interfaces for a conductor.

Parameters **conductor_id** Database ID of conductor.

Returns List of `ConductorHardwareInterfaces` objects.

list_hardware_type_interfaces(*hardware_types*)
List registered hardware interfaces for given hardware types.

This is restricted to only active conductors. :param hardware_types: list of hardware types to filter by. :returns: list of `ConductorHardwareInterfaces` objects.

node_tag_exists(*node_id*, *tag*)
Check if the specified tag exist on the node.

Parameters

- **node_id** The id of a node.
- **tag** A tag string.

Returns True if the tag exists otherwise False.

Raises NodeNotFound if the node is not found.

node_trait_exists(*node_id*, *trait*)
Check if the specified trait exists on the node.

Parameters

- **node_id** The id of a node.
- **trait** A trait string.

Returns True if the trait exists otherwise False.

Raises NodeNotFound if the node is not found.

query_node_history_records_for_purge(*conductor_id*)
Utility method to identify nodes to clean history records for.

Parameters **conductor_id** Id value for the conductor to perform this query on behalf of.

Returns A dictionary with key values of node database ID values and a list of values associated with the node.

register_conductor(*values*, *update_existing=False*)
Register an active conductor with the cluster.

Parameters

- **values** A dict of values which must contain the following:

```
{
 'hostname': the unique hostname which identifies
             this Conductor service.
 'drivers': a list of supported drivers.
 'version': the version of the object.Conductor
}
```

- **update_existing** When false, registration will raise an exception when a conflicting online record is found. When true, will overwrite the existing record. Default: False.

Returns A conductor.

Raises ConductorAlreadyRegistered

register_conductor_hardware_interfaces(*conductor_id*, *interfaces*)
Registers hardware interfaces for a conductor.

Parameters

- **conductor_id** Database ID of conductor to register for.
- **hardware_type** Name of hardware type for the interfaces.
- **interface_type** Type of interfaces, e.g. deploy or boot.
- **interfaces** List of interface names to register.
- **default_interface** String, the default interface for this hardware type and interface type.

Raises ConductorHardwareInterfacesAlreadyRegistered if at least one of the interfaces in the combination of all parameters is already registered.

release_node(*tag*, *node_id*)
Release the reservation on a node.

Parameters

- **tag** A string uniquely identifying the reservation holder.
- **node_id** A node id or uuid.

Raises NodeNotFound if the node is not found.

Raises NodeLocked if the node is reserved by another host.

Raises NodeNotLocked if the node was found to not have a reservation at all.

reserve_node(*tag*, *node_id*)
Reserve a node.

To prevent other ManagerServices from manipulating the given Node while a Task is performed, mark it reserved by this host.

Parameters

- **tag** A string uniquely identifying the reservation holder.
- **node_id** A node id or uuid.

Returns A Node object.

Raises NodeNotFound if the node is not found.

Raises NodeLocked if the node is already reserved.

set_node_tags(*node_id*, *tags*)
Replace all of the node tags with specified list of tags.

This ignores duplicate tags in the specified list.

Parameters

- **node_id** The id of a node.
- **tags** List of tags.

Returns A list of NodeTag objects.

Raises NodeNotFound if the node is not found.

set_node_traits(*node_id*, *traits*, *version*)
Replace all of the node traits with specified list of traits.

This ignores duplicate traits in the specified list.

Parameters

- **node_id** The id of a node.
- **traits** List of traits.
- **version** the version of the object.Trait.

Returns A list of NodeTrait objects.

Raises InvalidParameterValue if setting the traits would exceed the per-node traits limit.

Raises NodeNotFound if the node is not found.

take_over_allocation(*allocation_id*, *old_conductor_id*, *new_conductor_id*)
Do a take over for an allocation.

The allocation is only updated if the old conductor matches the provided value, thus guarding against races.

Parameters

- **allocation_id** Allocation ID
- **old_conductor_id** The conductor ID we expect to be the current `conductor_affinity` of the allocation.
- **new_conductor_id** The conductor ID of the new `conductor_affinity`.

Returns True if the take over was successful, False otherwise.

Raises AllocationNotFound

touch_conductor(*hostname*)
Mark a conductor as active by updating its updated_at property.

Parameters **hostname** The hostname of this conductor service.

Raises ConductorNotFound

touch_node_provisioning(*node_id*)
Mark the nodes provisioning as running.

Mark the nodes provisioning as running by updating its provision_updated_at property.

Parameters **node_id** The id of a node.

Raises NodeNotFound

unregister_conductor(*hostname*)
Remove this conductor from the service registry immediately.

Parameters **hostname** The hostname of this conductor service.

Raises ConductorNotFound

unregister_conductor_hardware_interfaces(*conductor_id*)
Unregisters all hardware interfaces for a conductor.

Parameters **conductor_id** Database ID of conductor to unregister for.

unset_node_tags(*node_id*)
Remove all tags of the node.

Parameters **node_id** The id of a node.

Raises NodeNotFound if the node is not found.

unset_node_traits(*node_id*)
Remove all traits of the node.

Parameters **node_id** The id of a node.

Raises NodeNotFound if the node is not found.

update_allocation(*allocation_id*, *values*, *update_node=True*)
Update properties of an allocation.

Parameters

- **allocation_id** Allocation ID
- **values** Dict of values to update.
- **update_node** If True and node_id is updated, update the node with instance_uuid and traits from the allocation

Returns An allocation.

Raises AllocationNotFound

Raises AllocationDuplicateName

Raises InstanceAssociated

Raises NodeAssociated

update_bios_setting_list(*node_id*, *settings*, *version*)
Update a list of BIOSSetting records.

Parameters

- **node_id** The node id.
- **settings** A list of BIOS Settings to be updated.

```
[
  {
   'name': String,
   'value': String,
   additional settings from BIOS registry
  },
  {
   'name': String,
   'value': String,
   additional settings from BIOS registry
  },
  ...
]
```

- **version** the version of the object.BIOSSetting.

Returns A list of BIOSSetting objects.

Raises NodeNotFound if the node is not found.

Raises BIOSSettingNotFound if any of the settings is not found.

update_chassis(*chassis_id*, *values*)
Update properties of an chassis.

Parameters

- **chassis_id** The id or the uuid of a chassis.
- **values** Dict of values to update.

Returns A chassis.

update_deploy_template(*template_id*, *values*)
Update a deployment template.

Parameters

- **template_id** ID of the deployment template to update.
- **values** A dict describing the deployment template. For example:

```
{
 'uuid': uuidutils.generate_uuid(),
 'name': 'CUSTOM_DT1',
}
```

Raises DeployTemplateDuplicateName if a deploy template with the same name exists.

Raises DeployTemplateNotFound if the deploy template does not exist.

Returns A deploy template.

update_node(*node_id*, *values*)
Update properties of a node.

Parameters

- **node_id** The id or uuid of a node.
- **values** Dict of values to update. May be a partial list, eg. when setting the properties for a driver. For example:

```
{
 'driver_info':
     {
      'my-field-1': val1,
      'my-field-2': val2,
     }
}
```

Returns A node.

Raises NodeAssociated

Raises NodeNotFound

update_port(*port_id*, *values*)
Update properties of an port.

Parameters

- **port_id** The id or MAC of a port.
- **values** Dict of values to update.

Returns A port.

update_portgroup(*portgroup_id*, *values*)
Update properties of a portgroup.

Parameters

- **portgroup_id** The UUID or MAC of a portgroup.
- **values** Dict of values to update. May contain the following keys: uuid name node_id address extra created_at updated_at

Returns A portgroup.

Raises InvalidParameterValue

Raises PortgroupNotFound

Raises PortgroupDuplicateName

Raises PortgroupMACAlreadyExists

update_to_latest_versions(*context*, *max_count*)
Updates objects to their latest known versions.

This scans all the tables and for objects that are not in their latest version, updates them to that version.

Parameters

- **context** the admin context
- **max_count** The maximum number of objects to migrate. Must be >= 0. If zero, all the objects will be migrated.

Returns A 2-tuple, 1. the total number of objects that need to be migrated (at the beginning of this call) and 2. the number of migrated objects.

update_volume_connector(*ident*, *connector_info*)
Update properties of a volume connector.

Parameters

- **ident** The UUID or integer ID of a volume connector.
- **connector_info** Dictionary containing the information about connector to update.

Returns A volume connector.

Raises VolumeConnectorTypeAndIdAlreadyExists If another connector already exists with a matching type and connector_id field.

Raises VolumeConnectorNotFound If a volume connector with the specified ident does not exist.

Raises InvalidParameterValue When a UUID is included in connector_info.

update_volume_target(*ident*, *target_info*)
Update information for a volume target.

Parameters

- **ident** The UUID or integer ID of a volume target.
- **target_info** Dictionary containing the information about volume target to update.

Returns A volume target.

Raises InvalidParameterValue if a UUID is included in target_info.

Raises VolumeTargetBootIndexAlreadyExists if a volume target already exists with the same boot index and node ID.

Raises VolumeTargetNotFound if no volume target with this ident exists.

ironic.db.sqlalchemy.api.**add_allocation_filter_by_conductor**(*query*, *value*)

ironic.db.sqlalchemy.api.**add_allocation_filter_by_node**(*query*, *value*)

ironic.db.sqlalchemy.api.**add_identity_filter**(*query*, *value*)
Adds an identity filter to a query.

Filters results by ID, if supplied value is a valid integer. Otherwise attempts to filter results by UUID.

Parameters

- **query** Initial query to add filter to.
- **value** Value for filtering results by.

Returns Modified query.

ironic.db.sqlalchemy.api.**add_node_filter_by_chassis**(*query*, *value*)

ironic.db.sqlalchemy.api.**add_port_filter**(*query*, *value*)
Adds a port-specific filter to a query.

Filters results by address, if supplied value is a valid MAC address. Otherwise attempts to filter results by identity.

Parameters

- **query** Initial query to add filter to.
- **value** Value for filtering results by.

Returns Modified query.

ironic.db.sqlalchemy.api.**add_port_filter_by_node**(*query*, *value*)

ironic.db.sqlalchemy.api.**add_port_filter_by_node_owner**(*query*, *value*)

ironic.db.sqlalchemy.api.**add_port_filter_by_node_project**(*query*, *value*)

ironic.db.sqlalchemy.api.**add_port_filter_by_portgroup**(*query*, *value*)

ironic.db.sqlalchemy.api.**add_portgroup_filter**(*query*, *value*)
Adds a portgroup-specific filter to a query.

Filters results by address, if supplied value is a valid MAC address. Otherwise attempts to filter results by identity.

Parameters

- **query** Initial query to add filter to.
- **value** Value for filtering results by.

Returns Modified query.

ironic.db.sqlalchemy.api.**add_portgroup_filter_by_node**(*query*, *value*)

ironic.db.sqlalchemy.api.**add_portgroup_filter_by_node_project**(*query*, *value*)

ironic.db.sqlalchemy.api.**add_volume_conn_filter_by_node_project**(*query*, *value*)

ironic.db.sqlalchemy.api.**add_volume_target_filter_by_node_project**(*query*, *value*)

ironic.db.sqlalchemy.api.**get_backend**()
The backend is this module itself.

ironic.db.sqlalchemy.api.**model_query**(*model*, **args*, ***kwargs*)
Query helper for simpler session usage.

Parameters **session** if present, the session to use

ironic.db.sqlalchemy.migration module

ironic.db.sqlalchemy.migration.**create_schema**(*config=None*, *engine=None*)
Create database schema from models description.

Can be used for initial installation instead of upgrade(head).

ironic.db.sqlalchemy.migration.**downgrade**(*revision*, *config=None*)
Used for downgrading database.

Parameters **version** (*string*) Desired database version

ironic.db.sqlalchemy.migration.**revision**(*message=None*, *autogenerate=False*, *config=None*)
Creates template for migration.

Parameters

- **message** (*string*) Text that will be used for migration title
- **autogenerate** (*bool*) If True - generates diff based on current database state

ironic.db.sqlalchemy.migration.**stamp**(*revision*, *config=None*)
Stamps database with provided revision.

Dont run any migrations.

Parameters **revision** (*string*) Should match one from repository or head - to stamp database with most recent revision

ironic.db.sqlalchemy.migration.**upgrade**(*revision*, *config=None*)
Used for upgrading database.

Parameters **version** (*string*) Desired database version

ironic.db.sqlalchemy.migration.**version**(*config=None*, *engine=None*)
Current database version.

Returns Database version

Return type string

ironic.db.sqlalchemy.models module

SQLAlchemy models for baremetal data.

class ironic.db.sqlalchemy.models.**Allocation**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Represents an allocation of a node for deployment.

candidate_nodes

conductor_affinity

created_at

extra

id

last_error

name

node_id

owner

resource_class

state

traits

updated_at

uuid

version

class ironic.db.sqlalchemy.models.**BIOSSetting**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Represents a bios setting of a bare metal node.

allowable_values

attribute_type

created_at

lower_bound

max_length

min_length

name

node_id

read_only

reset_required

unique

updated_at

upper_bound

value

version

class ironic.db.sqlalchemy.models.**Chassis**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Represents a hardware chassis.

created_at

description

extra

id

updated_at

uuid

version

class ironic.db.sqlalchemy.models.**Conductor**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Represents a conductor service entry.

conductor_group

created_at

drivers

hostname

id

online

updated_at

version

class ironic.db.sqlalchemy.models.**ConductorHardwareInterfaces**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Internal table used to track what is loaded on each conductor.

conductor_id

created_at

default

hardware_type

id

interface_name

interface_type

updated_at

version

class ironic.db.sqlalchemy.models.DeployTemplate(**kwargs)
Bases: sqlalchemy.orm.decl_api.Base

Represents a deployment template.

created_at

extra

id

name

updated_at

uuid

version

class ironic.db.sqlalchemy.models.DeployTemplateStep(**kwargs)
Bases: sqlalchemy.orm.decl_api.Base

Represents a deployment step in a deployment template.

args

created_at

deploy_template

deploy_template_id

id

interface

priority

step

updated_at

version

class ironic.db.sqlalchemy.models.IronicBase
Bases: oslo_db.sqlalchemy.models.TimestampMixin, oslo_db.sqlalchemy.models.ModelBase

as_dict()

metadata = None

version = Column(None, String(length=15), table=None)

class ironic.db.sqlalchemy.models.**Node**(***kwargs*)

Bases: sqlalchemy.orm.decl_api.Base

Represents a bare metal node.

allocation_id

automated_clean

bios_interface

boot_interface

boot_mode

chassis_id

clean_step

conductor_affinity

conductor_group

console_enabled

console_interface

created_at

deploy_interface

deploy_step

description

driver

driver_info

driver_internal_info

extra

fault

id

inspect_interface

inspection_finished_at

inspection_started_at

instance_info

instance_uuid

last_error

lessee

maintenance

maintenance_reason

management_interface

name

network_data

network_interface

owner

power_interface

power_state

properties

protected

protected_reason

provision_state

provision_updated_at

raid_config

raid_interface

rescue_interface

reservation

resource_class

retired

retired_reason

secure_boot

storage_interface

target_power_state

target_provision_state

target_raid_config

updated_at

uuid

vendor_interface

version

class ironic.db.sqlalchemy.models.**NodeHistory**(***kwargs*)

Bases: sqlalchemy.orm.decl_api.Base

Represents a history event of a bare metal node.

conductor

created_at

event

event_type

id

node_id

severity

updated_at

user

uuid

version

class ironic.db.sqlalchemy.models.**NodeTag**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Represents a tag of a bare metal node.

created_at

node

node_id

tag

updated_at

version

class ironic.db.sqlalchemy.models.**NodeTrait**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Represents a trait of a bare metal node.

created_at

node

node_id

trait

updated_at

version

class ironic.db.sqlalchemy.models.**Port**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Represents a network port of a bare metal node.

address

created_at

extra

id

internal_info

is_smartnic

local_link_connection

name

node_id

physical_network

portgroup_id

pxe_enabled

updated_at

uuid

version

class ironic.db.sqlalchemy.models.**Portgroup**(***kwargs*)

Bases: sqlalchemy.orm.decl_api.Base

Represents a group of network ports of a bare metal node.

address

created_at

extra

id

internal_info

mode

name

node_id

properties

standalone_ports_supported

updated_at

uuid

version

class ironic.db.sqlalchemy.models.**VolumeConnector**(***kwargs*)

Bases: sqlalchemy.orm.decl_api.Base

Represents a volume connector of a bare metal node.

connector_id

created_at

extra

id

node_id

type

updated_at

uuid

version

class ironic.db.sqlalchemy.models.**VolumeTarget**(***kwargs*)
Bases: sqlalchemy.orm.decl_api.Base

Represents a volume target of a bare metal node.

boot_index

created_at

extra

id

node_id

properties

updated_at

uuid

version

volume_id

volume_type

ironic.db.sqlalchemy.models.**get_class**(*model_name*)
Returns the model class with the specified name.

Parameters **model_name** the name of the class

Returns the class with the specified name

Raises Exception if there is no class associated with the name

ironic.db.sqlalchemy.models.**table_args**()

Module contents

Submodules

ironic.db.api module

Base classes for storage engines

class ironic.db.api.**Connection**
Bases: object

Base class for storage system connections.

abstract add_node_tag(*node_id*, *tag*)
Add tag to the node.

If the node_id and tag pair already exists, this should still succeed.

Parameters

- **node_id** The id of a node.
- **tag** A tag string.

Returns the NodeTag object.

Raises NodeNotFound if the node is not found.

abstract add_node_trait(*node_id*, *trait*, *version*)
Add trait to the node.

If the node_id and trait pair already exists, this should still succeed.

Parameters

- **node_id** The id of a node.
- **trait** A trait string.
- **version** the version of the object.Trait.

Returns the NodeTrait object.

Raises InvalidParameterValue if adding the trait would exceed the per-node traits limit.

Raises NodeNotFound if the node is not found.

abstract bulk_delete_node_history_records(*node_id*, *limit*)
Utility method to bulk delete node history entries.

Parameters entires A list of node history entriy ids to be queried for deletion.

abstract check_node_list(*idents*)
Check a list of node identities and map it to UUIDs.

This call takes a list of node names and/or UUIDs and tries to convert them to UUIDs. It fails early if any identities cannot possible be used as names or UUIDs.

Parameters idents List of identities.

Returns A mapping from requests identities to node UUIDs.

Raises NodeNotFound if some identities were not found or cannot be valid names or UUIDs.

abstract check_versions(*ignore_models=()*)
Checks the whole database for incompatible objects.

This scans all the tables in search of objects that are not supported; i.e., those that are not specified in *ironic.common.release_mappings.RELEASE_MAPPING*.

Parameters ignore_models List of model names to skip.

Returns A Boolean. True if all the objects have supported versions; False otherwise.

abstract create_allocation(*values*)
Create a new allocation.

Parameters values Dict of values to create an allocation with

Returns An allocation

Raises AllocationDuplicateName

Raises AllocationAlreadyExists

abstract create_bios_setting_list(*node_id*, *settings*, *version*)
Create a list of BIOSSetting records for a given node.

Parameters

- **node_id** The node id.
- **settings** A list of BIOS Settings to be created.

```
[
  {
   'name': String,
   'value': String,
   additional settings from BIOS registry
  },
  {
   'name': String,
   'value': String,
   additional settings from BIOS registry
  },
  ...
]
```

- **version** the version of the object.BIOSSetting.

Returns A list of BIOSSetting object.

Raises NodeNotFound if the node is not found.

Raises BIOSSettingAlreadyExists if any of the setting records already exists.

abstract create_chassis(*values*)
Create a new chassis.

Parameters **values** Dict of values.

abstract create_deploy_template(*values*)
Create a deployment template.

Parameters **values** A dict describing the deployment template. For example:

```
{
 'uuid': uuidutils.generate_uuid(),
 'name': 'CUSTOM_DT1',
}
```

Raises DeployTemplateDuplicateName if a deploy template with the same name exists.

Raises DeployTemplateAlreadyExists if a deploy template with the same UUID exists.

Returns A deploy template.

abstract create_node(*values*)
Create a new node.

Parameters **values** A dict containing several items used to identify and track the node, and several dicts which are passed into the Drivers when managing this node. For example:

```
{
 'uuid': uuidutils.generate_uuid(),
 'instance_uuid': None,
 'power_state': states.POWER_OFF,
 'provision_state': states.AVAILABLE,
 'driver': 'ipmi',
 'driver_info': { ... },
 'properties': { ... },
 'extra': { ... },
}
```

Raises InvalidParameterValue if values contains tags or traits.

Returns A node.

abstract create_node_history(*values*)
Create a new history record.

Parameters **values** Dict of values.

abstract create_port(*values*)
Create a new port.

Parameters **values** Dict of values.

abstract create_portgroup(*values*)
Create a new portgroup.

Parameters **values** Dict of values with the following keys: id uuid name node_id address extra created_at updated_at

Returns A portgroup

Raises PortgroupDuplicateName

Raises PortgroupMACAlreadyExists

Raises PortgroupAlreadyExists

abstract create_volume_connector(*connector_info*)
Create a new volume connector.

Parameters **connector_info** Dictionary containing information about the connector. Example:

```
{
    'uuid': '000000-..',
    'type': 'wwnn',
    'connector_id': '00:01:02:03:04:05:06',
    'node_id': 2
}
```

Returns A volume connector.

Raises VolumeConnectorTypeAndIdAlreadyExists If a connector already exists with a matching type and connector_id.

Raises VolumeConnectorAlreadyExists If a volume connector with the same UUID already exists.

abstract create_volume_target(*target_info*)
Create a new volume target.

Parameters **target_info** Dictionary containing the information about the volume target. Example:

```
{
    'uuid': '000000-..',
    'node_id': 2,
    'boot_index': 0,
    'volume_id': '12345678-...'
    'volume_type': 'some type',
}
```

Returns A volume target.

Raises VolumeTargetBootIndexAlreadyExists if a volume target already exists with the same boot index and node ID.

Raises VolumeTargetAlreadyExists if a volume target with the same UUID exists.

abstract delete_bios_setting_list(*node_id*, *names*)
Delete a list of BIOS settings.

Parameters

- **node_id** The node id.
- **names** List of BIOS setting names to be deleted.

Raises NodeNotFound if the node is not found.

Raises BIOSSettingNotFound if any of BIOS setting name is not found.

abstract delete_node_tag(*node_id*, *tag*)
Delete specified tag from the node.

Parameters

- **node_id** The id of a node.
- **tag** A tag string.

Raises NodeNotFound if the node is not found.

Raises NodeTagNotFound if the tag is not found.

abstract delete_node_trait(*node_id*, *trait*)
Delete specified trait from the node.

Parameters

- **node_id** The id of a node.
- **trait** A trait string.

Raises NodeNotFound if the node is not found.

Raises NodeTraitNotFound if the trait is not found.

abstract destroy_allocation(*allocation_id*)
Destroy an allocation.

Parameters **allocation_id** Allocation ID

Raises AllocationNotFound

abstract destroy_chassis(*chassis_id*)
Destroy a chassis.

Parameters **chassis_id** The id or the uuid of a chassis.

abstract destroy_deploy_template(*template_id*)
Destroy a deployment template.

Parameters **template_id** ID of the deployment template to destroy.

Raises DeployTemplateNotFound if the deploy template does not exist.

abstract destroy_node(*node_id*)
Destroy a node and its associated resources.

Destroy a node, including any associated ports, port groups, tags, traits, volume connectors, and volume targets.

Parameters **node_id** The ID or UUID of a node.

abstract destroy_node_history_by_uuid(*history_uuid*)
Destroy a history record.

Parameters **history_uuid** The uuid of a history record

abstract destroy_port(*port_id*)
Destroy an port.

Parameters **port_id** The id or MAC of a port.

abstract destroy_portgroup(*portgroup_id*)
Destroy a portgroup.

Parameters **portgroup_id** The UUID or MAC of a portgroup.

Raises PortgroupNotEmpty

Raises PortgroupNotFound

abstract destroy_volume_connector(*ident*)
Destroy a volume connector.

Parameters **ident** The UUID or integer ID of a volume connector.

Raises VolumeConnectorNotFound If a volume connector with the specified ident does not exist.

abstract destroy_volume_target(*ident*)
Destroy a volume target.

Parameters **ident** The UUID or integer ID of a volume target.

Raises VolumeTargetNotFound if a volume target with the specified ident does not exist.

abstract get_active_hardware_type_dict(*use_groups=False*)

Retrieve hardware types for the registered and active conductors.

Parameters **use_groups** Whether to factor conductor_group into the keys.

Returns

A dict which maps hardware type names to the set of hosts which support them. For example:

```
{hardware-type-a: set([host1, host2]),
 hardware-type-b: set([host2, host3])}
```

abstract get_allocation_by_id(*allocation_id*)

Return an allocation representation.

Parameters **allocation_id** The id of an allocation.

Returns An allocation.

Raises AllocationNotFound

abstract get_allocation_by_name(*name*)

Return an allocation representation.

Parameters **name** The logical name of an allocation.

Returns An allocation.

Raises AllocationNotFound

abstract get_allocation_by_uuid(*allocation_uuid*)

Return an allocation representation.

Parameters **allocation_uuid** The uuid of an allocation.

Returns An allocation.

Raises AllocationNotFound

abstract get_allocation_list(*filters=None*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)

Return a list of allocations.

Parameters

- **filters** Filters to apply. Defaults to None.

  node_uuid uuid of node

  state allocation state

  resource_class requested resource class

- **limit** Maximum number of allocations to return.
- **marker** The last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** Direction in which results should be sorted. (asc, desc)

Returns A list of allocations.

abstract get_bios_setting(*node_id*, *name*)
Retrieve BIOS setting value.

Parameters

- **node_id** The node id.
- **name** String containing name of BIOS setting to be retrieved.

Returns The BIOSSetting object.

Raises NodeNotFound if the node is not found.

Raises BIOSSettingNotFound if the BIOS setting is not found.

abstract get_bios_setting_list(*node_id*)
Retrieve BIOS settings of a given node.

Parameters **node_id** The node id.

Returns A list of BIOSSetting objects.

Raises NodeNotFound if the node is not found.

abstract get_chassis_by_id(*chassis_id*)
Return a chassis representation.

Parameters **chassis_id** The id of a chassis.

Returns A chassis.

abstract get_chassis_by_uuid(*chassis_uuid*)
Return a chassis representation.

Parameters **chassis_uuid** The uuid of a chassis.

Returns A chassis.

abstract get_chassis_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)
Return a list of chassis.

Parameters

- **limit** Maximum number of chassis to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** direction in which results should be sorted. (asc, desc)

abstract get_conductor(*hostname*, *online=True*)
Retrieve a conductors service record from the database.

Parameters

- **hostname** The hostname of the conductor service.
- **online** Specify the filter value on the *online* field when querying conductors. The `online` field is ignored if this value is set to None.

Returns A conductor.

Raises ConductorNotFound if the conductor with given hostname does not exist or doesnt meet the specified online expectation.

abstract get_conductor_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)

Return a list of conductors.

Parameters

- **limit** Maximum number of conductors to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** direction in which results should be sorted. (asc, desc)

abstract get_deploy_template_by_id(*template_id*)

Retrieve a deployment template by ID.

Parameters **template_id** ID of the deployment template to retrieve.

Raises DeployTemplateNotFound if the deploy template does not exist.

Returns A deploy template.

abstract get_deploy_template_by_name(*template_name*)

Retrieve a deployment template by name.

Parameters **template_name** name of the deployment template to retrieve.

Raises DeployTemplateNotFound if the deploy template does not exist.

Returns A deploy template.

abstract get_deploy_template_by_uuid(*template_uuid*)

Retrieve a deployment template by UUID.

Parameters **template_uuid** UUID of the deployment template to retrieve.

Raises DeployTemplateNotFound if the deploy template does not exist.

Returns A deploy template.

abstract get_deploy_template_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)

Retrieve a list of deployment templates.

Parameters

- **limit** Maximum number of deploy templates to return.
- **marker** The last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** Direction in which results should be sorted. (asc, desc)

Returns A list of deploy templates.

abstract get_deploy_template_list_by_names(*names*)

Return a list of deployment templates with one of a list of names.

Parameters **names** List of names to filter by.

Returns A list of deploy templates.

abstract get_node_by_id(*node_id*)
Return a node.

Parameters **node_id** The id of a node.

Returns A node.

abstract get_node_by_instance(*instance*)
Return a node.

Parameters **instance** The instance uuid to search for.

Returns A node.

Raises InstanceNotFound if the instance is not found.

Raises InvalidUUID if the instance uuid is invalid.

abstract get_node_by_name(*node_name*)
Return a node.

Parameters **node_name** The logical name of a node.

Returns A node.

abstract get_node_by_port_addresses(*addresses*)
Find a node by any matching port address.

Parameters **addresses** list of port addresses (e.g. MACs).

Returns Node object.

Raises NodeNotFound if none or several nodes are found.

abstract get_node_by_uuid(*node_uuid*)
Return a node.

Parameters **node_uuid** The uuid of a node.

Returns A node.

abstract get_node_history_by_id(*history_id*)
Return a node history representation.

Parameters **history_id** The id of a history record.

Returns A history.

abstract get_node_history_by_node_id(*node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)
List all the history records for a given node.

Parameters

- **node_id** The integer node ID.
- **limit** Maximum number of history records to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted
- **sort_dir** direction in which results should be sorted (asc, desc)

Returns A list of histories.

abstract get_node_history_by_uuid(*history_uuid*)
Return a node history representation.

Parameters **history_uuid** The uuid of a history record

Returns A history.

abstract get_node_history_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)
Return a list of node history records

Parameters

- **limit** Maximum number of history records to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** direction in which results should be sorted. (asc, desc)

abstract get_node_list(*filters=None*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *fields=None*)
Return a list of nodes.

Parameters

- **filters** Filters to apply. Defaults to None.

  associated True | False

  reserved True | False

  maintenance True | False

  chassis_uuid uuid of chassis

  driver drivers name

  provision_state provision state of node

  provisioned_before nodes with provision_updated_at field before this interval in seconds

- **limit** Maximum number of nodes to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** direction in which results should be sorted. (asc, desc)
- **fields** Comma separated field list to return, to allow for only specific fields to be returned to have maximum API performance calls where not all columns are needed from the database.

abstract get_node_tags_by_node_id(*node_id*)
Get node tags based on its id.

Parameters **node_id** The id of a node.

Returns A list of NodeTag objects.

Raises NodeNotFound if the node is not found.

abstract get_node_traits_by_node_id(*node_id*)

Get node traits based on its id.

Parameters **node_id** The id of a node.

Returns A list of NodeTrait objects.

Raises NodeNotFound if the node is not found.

abstract get_nodeinfo_list(*columns=None*, *filters=None*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)

Get specific columns for matching nodes.

Return a list of the specified columns for all nodes that match the specified filters.

Parameters

- **columns** List of column names to return. Defaults to id column when columns == None.
- **filters** Filters to apply. Defaults to None.

  associated True | False

  chassis_uuid uuid of chassis

  conductor_group conductor group name

  console_enabled True | False

  description_contains substring in description

  driver drivers name

  fault current fault type

  id numeric ID

  inspection_started_before nodes with inspection_started_at field before this interval in seconds

  instance_uuid uuid of instance

  lessee nodes lessee (e.g. project ID)

  maintenance True | False

  owner nodes owner (e.g. project ID)

  project either owner or lessee

  reserved True | False

  reserved_by_any_of [conductor1, conductor2]

  resource_class resource class name

  retired True | False

  provision_state provision state of node

  provision_state_in provision state of node (multiple possibilities)

provisioned_before nodes with provision_updated_at field before this interval in seconds

uuid uuid of node

uuid_in uuid of node (multiple possibilities)

with_power_state True | False

- **limit** Maximum number of nodes to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** direction in which results should be sorted. (asc, desc)

Returns A list of tuples of the specified columns.

abstract get_not_versions(*model_name*, *versions*)
Returns objects with versions that are not the specified versions.

Parameters

- **model_name** the name of the model (class) of desired objects
- **versions** list of versions of objects not to be returned

Returns list of the DB objects

Raises IronicException if there is no class associated with the name

abstract get_offline_conductors(*field='hostname'*)
Get a list conductors that are offline (dead).

Parameters **field** A field to return, hostname by default.

Returns A list of requested fields of offline conductors.

abstract get_online_conductors()
Get a list conductor hostnames that are online and active.

Returns A list of conductor hostnames.

abstract get_port_by_address(*address*)
Return a network port representation.

Parameters **address** The MAC address of a port.

Returns A port.

abstract get_port_by_id(*port_id*)
Return a network port representation.

Parameters **port_id** The id of a port.

Returns A port.

abstract get_port_by_name(*port_name*)
Return a network port representation.

Parameters **port_name** The name of a port.

Returns A port.

abstract get_port_by_uuid(*port_uuid*)
Return a network port representation.

Parameters **port_uuid** The uuid of a port.

Returns A port.

abstract get_port_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)
Return a list of ports.

Parameters

- **limit** Maximum number of ports to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** direction in which results should be sorted. (asc, desc)

abstract get_portgroup_by_address(*address*, *project=None*)
Return a network portgroup representation.

Parameters

- **address** The MAC address of a portgroup.
- **project** A node owner or lessee to filter by.

Returns A portgroup.

Raises PortgroupNotFound

abstract get_portgroup_by_id(*portgroup_id*)
Return a network portgroup representation.

Parameters **portgroup_id** The id of a portgroup.

Returns A portgroup.

Raises PortgroupNotFound

abstract get_portgroup_by_name(*name*)
Return a network portgroup representation.

Parameters **name** The logical name of a portgroup.

Returns A portgroup.

Raises PortgroupNotFound

abstract get_portgroup_by_uuid(*portgroup_uuid*)
Return a network portgroup representation.

Parameters **portgroup_uuid** The uuid of a portgroup.

Returns A portgroup.

Raises PortgroupNotFound

abstract get_portgroup_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)
Return a list of portgroups.

Parameters

- **limit** Maximum number of portgroups to return.
- **marker** The last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** Direction in which results should be sorted. (asc, desc)
- **project** A node owner or lessee to filter by.

Returns A list of portgroups.

abstract get_portgroups_by_node_id(*node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)

List all the portgroups for a given node.

Parameters

- **node_id** The integer node ID.
- **limit** Maximum number of portgroups to return.
- **marker** The last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted
- **sort_dir** Direction in which results should be sorted (asc, desc)
- **project** A node owner or lessee to filter by.

Returns A list of portgroups.

abstract get_ports_by_node_id(*node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)

List all the ports for a given node.

Parameters

- **node_id** The integer node ID.
- **limit** Maximum number of ports to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted
- **sort_dir** direction in which results should be sorted (asc, desc)

Returns A list of ports.

abstract get_ports_by_portgroup_id(*portgroup_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)

List all the ports for a given portgroup.

Parameters

- **portgroup_id** The integer portgroup ID.
- **limit** Maximum number of ports to return.
- **marker** The last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted
- **sort_dir** Direction in which results should be sorted (asc, desc)

Returns A list of ports.

abstract get_volume_connector_by_id(*db_id*)
Return a volume connector representation.

Parameters **db_id** The integer database ID of a volume connector.

Returns A volume connector with the specified ID.

Raises VolumeConnectorNotFound If a volume connector with the specified ID is not found.

abstract get_volume_connector_by_uuid(*connector_uuid*)
Return a volume connector representation.

Parameters **connector_uuid** The UUID of a connector.

Returns A volume connector with the specified UUID.

Raises VolumeConnectorNotFound If a volume connector with the specified UUID is not found.

abstract get_volume_connector_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)
Return a list of volume connectors.

Parameters

- **limit** Maximum number of volume connectors to return.
- **marker** The last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** Direction in which results should be sorted. (asc, desc)
- **project** The associated node project to search with.

Returns a list of `VolumeConnector` objects

Returns A list of volume connectors.

Raises InvalidParameterValue If sort_key does not exist.

abstract get_volume_connectors_by_node_id(*node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)
List all the volume connectors for a given node.

Parameters

- **node_id** The integer node ID.
- **limit** Maximum number of volume connectors to return.
- **marker** The last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted
- **sort_dir** Direction in which results should be sorted (asc, desc)
- **project** The associated node project to search with.

Returns a list of `VolumeConnector` objects

Returns A list of volume connectors.

Raises InvalidParameterValue If sort_key does not exist.

abstract get_volume_target_by_id(*db_id*)
Return a volume target representation.

Parameters **db_id** The database primary key (integer) ID of a volume target.

Returns A volume target.

Raises VolumeTargetNotFound if no volume target with this ID exists.

abstract get_volume_target_by_uuid(*uuid*)
Return a volume target representation.

Parameters **uuid** The UUID of a volume target.

Returns A volume target.

Raises VolumeTargetNotFound if no volume target with this UUID exists.

abstract get_volume_target_list(*limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)
Return a list of volume targets.

Parameters

- **limit** Maximum number of volume targets to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted.
- **sort_dir** direction in which results should be sorted. (asc, desc)
- **project** The associated node project to search with.

Returns a list of `VolumeConnector` objects

Returns A list of volume targets.

Raises InvalidParameterValue if sort_key does not exist.

abstract get_volume_targets_by_node_id(*node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)
List all the volume targets for a given node.

Parameters

- **node_id** The integer node ID.
- **limit** Maximum number of volume targets to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted
- **sort_dir** direction in which results should be sorted (asc, desc)
- **project** The associated node project to search with.

Returns a list of `VolumeConnector` objects

Returns A list of volume targets.

Raises InvalidParameterValue if sort_key does not exist.

abstract get_volume_targets_by_volume_id(*volume_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)

List all the volume targets for a given volume id.

Parameters

- **volume_id** The UUID of the volume.
- **limit** Maximum number of volume targets to return.
- **marker** the last item of the previous page; we return the next result set.
- **sort_key** Attribute by which results should be sorted
- **sort_dir** direction in which results should be sorted (asc, desc)

Returns A list of volume targets.

Raises InvalidParameterValue if sort_key does not exist.

abstract list_conductor_hardware_interfaces(*conductor_id*)

List all registered hardware interfaces for a conductor.

Parameters **conductor_id** Database ID of conductor.

Returns List of `ConductorHardwareInterfaces` objects.

abstract list_hardware_type_interfaces(*hardware_types*)

List registered hardware interfaces for given hardware types.

This is restricted to only active conductors. :param hardware_types: list of hardware types to filter by. :returns: list of `ConductorHardwareInterfaces` objects.

abstract node_tag_exists(*node_id*, *tag*)

Check if the specified tag exist on the node.

Parameters

- **node_id** The id of a node.
- **tag** A tag string.

Returns True if the tag exists otherwise False.

Raises NodeNotFound if the node is not found.

abstract node_trait_exists(*node_id*, *trait*)

Check if the specified trait exists on the node.

Parameters

- **node_id** The id of a node.
- **trait** A trait string.

Returns True if the trait exists otherwise False.

Raises NodeNotFound if the node is not found.

abstract query_node_history_records_for_purge(*conductor_id*)

Utility method to identify nodes to clean history records for.

Parameters **conductor_id** Id value for the conductor to perform this query on behalf of.

Returns A dictionary with key values of node database ID values and a list of values associated with the node.

abstract register_conductor(*values*, *update_existing=False*)

Register an active conductor with the cluster.

Parameters

- **values** A dict of values which must contain the following:

```
{
 'hostname': the unique hostname which identifies
             this Conductor service.
 'drivers': a list of supported drivers.
 'version': the version of the object.Conductor
}
```

- **update_existing** When false, registration will raise an exception when a conflicting online record is found. When true, will overwrite the existing record. Default: False.

Returns A conductor.

Raises ConductorAlreadyRegistered

abstract register_conductor_hardware_interfaces(*conductor_id*, *hardware_type*, *interface_type*, *interfaces*, *default_interface*)

Registers hardware interfaces for a conductor.

Parameters

- **conductor_id** Database ID of conductor to register for.
- **hardware_type** Name of hardware type for the interfaces.
- **interface_type** Type of interfaces, e.g. deploy or boot.
- **interfaces** List of interface names to register.
- **default_interface** String, the default interface for this hardware type and interface type.

Raises ConductorHardwareInterfacesAlreadyRegistered if at least one of the interfaces in the combination of all parameters is already registered.

abstract release_node(*tag*, *node_id*)

Release the reservation on a node.

Parameters

- **tag** A string uniquely identifying the reservation holder.
- **node_id** A node id or uuid.

Raises NodeNotFound if the node is not found.

Raises NodeLocked if the node is reserved by another host.

Raises NodeNotLocked if the node was found to not have a reservation at all.

abstract reserve_node(*tag*, *node_id*)
Reserve a node.

To prevent other ManagerServices from manipulating the given Node while a Task is performed, mark it reserved by this host.

Parameters

- **tag** A string uniquely identifying the reservation holder.
- **node_id** A node id or uuid.

Returns A Node object.

Raises NodeNotFound if the node is not found.

Raises NodeLocked if the node is already reserved.

abstract set_node_tags(*node_id*, *tags*)
Replace all of the node tags with specified list of tags.

This ignores duplicate tags in the specified list.

Parameters

- **node_id** The id of a node.
- **tags** List of tags.

Returns A list of NodeTag objects.

Raises NodeNotFound if the node is not found.

abstract set_node_traits(*node_id*, *traits*, *version*)
Replace all of the node traits with specified list of traits.

This ignores duplicate traits in the specified list.

Parameters

- **node_id** The id of a node.
- **traits** List of traits.
- **version** the version of the object.Trait.

Returns A list of NodeTrait objects.

Raises InvalidParameterValue if setting the traits would exceed the per-node traits limit.

Raises NodeNotFound if the node is not found.

abstract take_over_allocation(*allocation_id*, *old_conductor_id*, *new_conductor_id*)
Do a take over for an allocation.

The allocation is only updated if the old conductor matches the provided value, thus guarding against races.

Parameters

- **allocation_id** Allocation ID

- **old_conductor_id** The conductor ID we expect to be the current `conductor_affinity` of the allocation.
- **new_conductor_id** The conductor ID of the new `conductor_affinity`.

Returns True if the take over was successful, False otherwise.

Raises AllocationNotFound

abstract touch_conductor(*hostname*)
Mark a conductor as active by updating its updated_at property.

Parameters **hostname** The hostname of this conductor service.

Raises ConductorNotFound

abstract touch_node_provisioning(*node_id*)
Mark the nodes provisioning as running.

Mark the nodes provisioning as running by updating its provision_updated_at property.

Parameters **node_id** The id of a node.

Raises NodeNotFound

abstract unregister_conductor(*hostname*)
Remove this conductor from the service registry immediately.

Parameters **hostname** The hostname of this conductor service.

Raises ConductorNotFound

abstract unregister_conductor_hardware_interfaces(*conductor_id*)
Unregisters all hardware interfaces for a conductor.

Parameters **conductor_id** Database ID of conductor to unregister for.

abstract unset_node_tags(*node_id*)
Remove all tags of the node.

Parameters **node_id** The id of a node.

Raises NodeNotFound if the node is not found.

abstract unset_node_traits(*node_id*)
Remove all traits of the node.

Parameters **node_id** The id of a node.

Raises NodeNotFound if the node is not found.

abstract update_allocation(*allocation_id*, *values*, *update_node=True*)
Update properties of an allocation.

Parameters

- **allocation_id** Allocation ID
- **values** Dict of values to update.
- **update_node** If True and node_id is updated, update the node with instance_uuid and traits from the allocation

Returns An allocation.

Raises AllocationNotFound

Raises AllocationDuplicateName

Raises InstanceAssociated

Raises NodeAssociated

abstract update_bios_setting_list(*node_id*, *settings*, *version*)
Update a list of BIOSSetting records.

Parameters

- **node_id** The node id.
- **settings** A list of BIOS Settings to be updated.

```
[
  {
   'name': String,
   'value': String,
   additional settings from BIOS registry
  },
  {
   'name': String,
   'value': String,
   additional settings from BIOS registry
  },
  ...
]
```

- **version** the version of the object.BIOSSetting.

Returns A list of BIOSSetting objects.

Raises NodeNotFound if the node is not found.

Raises BIOSSettingNotFound if any of the settings is not found.

abstract update_chassis(*chassis_id*, *values*)
Update properties of an chassis.

Parameters

- **chassis_id** The id or the uuid of a chassis.
- **values** Dict of values to update.

Returns A chassis.

abstract update_deploy_template(*template_id*, *values*)
Update a deployment template.

Parameters

- **template_id** ID of the deployment template to update.
- **values** A dict describing the deployment template. For example:

```
{
 'uuid': uuidutils.generate_uuid(),
 'name': 'CUSTOM_DT1',
}
```

Raises DeployTemplateDuplicateName if a deploy template with the same name exists.

Raises DeployTemplateNotFound if the deploy template does not exist.

Returns A deploy template.

abstract update_node(*node_id*, *values*)

Update properties of a node.

Parameters

- **node_id** The id or uuid of a node.
- **values** Dict of values to update. May be a partial list, eg. when setting the properties for a driver. For example:

```
{
 'driver_info':
     {
      'my-field-1': val1,
      'my-field-2': val2,
     }
}
```

Returns A node.

Raises NodeAssociated

Raises NodeNotFound

abstract update_port(*port_id*, *values*)

Update properties of an port.

Parameters

- **port_id** The id or MAC of a port.
- **values** Dict of values to update.

Returns A port.

abstract update_portgroup(*portgroup_id*, *values*)

Update properties of a portgroup.

Parameters

- **portgroup_id** The UUID or MAC of a portgroup.
- **values** Dict of values to update. May contain the following keys: uuid name node_id address extra created_at updated_at

Returns A portgroup.

Raises InvalidParameterValue

Raises PortgroupNotFound

Raises PortgroupDuplicateName

Raises PortgroupMACAlreadyExists

abstract update_to_latest_versions(*context*, *max_count*)
Updates objects to their latest known versions.

This scans all the tables and for objects that are not in their latest version, updates them to that version.

Parameters

- **context** the admin context
- **max_count** The maximum number of objects to migrate. Must be >= 0. If zero, all the objects will be migrated.

Returns A 2-tuple, 1. the total number of objects that need to be migrated (at the beginning of this call) and 2. the number of migrated objects.

abstract update_volume_connector(*ident*, *connector_info*)
Update properties of a volume connector.

Parameters

- **ident** The UUID or integer ID of a volume connector.
- **connector_info** Dictionary containing the information about connector to update.

Returns A volume connector.

Raises VolumeConnectorTypeAndIdAlreadyExists If another connector already exists with a matching type and connector_id field.

Raises VolumeConnectorNotFound If a volume connector with the specified ident does not exist.

Raises InvalidParameterValue When a UUID is included in connector_info.

abstract update_volume_target(*ident*, *target_info*)
Update information for a volume target.

Parameters

- **ident** The UUID or integer ID of a volume target.
- **target_info** Dictionary containing the information about volume target to update.

Returns A volume target.

Raises InvalidParameterValue if a UUID is included in target_info.

Raises VolumeTargetBootIndexAlreadyExists if a volume target already exists with the same boot index and node ID.

Raises VolumeTargetNotFound if no volume target with this ident exists.

ironic.db.api.**get_instance**()
Return a DB API instance.

ironic.db.migration module

Database setup and migration commands.

ironic.db.migration.**create_schema**()

ironic.db.migration.**get_backend**()

ironic.db.migration.**revision**(*message*, *autogenerate*)

ironic.db.migration.**stamp**(*version*)

ironic.db.migration.**upgrade**(*version=None*)
Migrate the database to *version* or the most recent version.

ironic.db.migration.**version**()

Module contents

ironic.dhcp package

Submodules

ironic.dhcp.base module

Abstract base class for dhcp providers.

class ironic.dhcp.base.**BaseDHCP**
Bases: object

Base class for DHCP provider APIs.

clean_dhcp_opts(*task*)
Clean up the DHCP BOOT options for all ports in *task*.

Parameters **task** A TaskManager instance.

Raises FailedToCleanDHCPOpts

get_ip_addresses(*task*)
Get IP addresses for all ports/portgroups in *task*.

Parameters **task** A TaskManager instance.

Returns List of IP addresses associated with tasks ports and portgroups.

abstract update_dhcp_opts(*task*, *options*, *vifs=None*)
Send or update the DHCP BOOT options for this node.

Parameters

- **task** A TaskManager instance.
- **options** this will be a list of dicts, e.g.

```
[{'opt_name': '67',
  'opt_value': 'pxelinux.0',
  'ip_version': 4},
```

(continues on next page)

(continued from previous page)

```
{'opt_name': '66',
 'opt_value': '123.123.123.456',
 'ip_version': 4}]
```

- **vifs** A dict with keys ports and portgroups and dicts as values. Each dict has key/value pairs of the form <ironic UUID>:<neutron port UUID>. e.g.

```
{'ports': {'port.uuid': vif.id},
 'portgroups': {'portgroup.uuid': vif.id}}
```

If the value is None, will get the list of ports/portgroups from the Ironic port/portgroup objects.

Raises FailedToUpdateDHCPOptOnPort

abstract update_port_dhcp_opts(*port_id*, *dhcp_options*, *token=None*, *context=None*)

Update one or more DHCP options on the specified port.

Parameters

- **port_id** designate which port these attributes will be applied to.
- **dhcp_options** this will be a list of dicts, e.g.

```
[{'opt_name': '67',
  'opt_value': 'pxelinux.0',
  'ip_version': 4},
 {'opt_name': '66',
  'opt_value': '123.123.123.456',
  'ip_version': 4}]
```

- **token** An optional authentication token. Deprecated, use context
- **context** (ironic.common.context.RequestContext) request context

Raises FailedToUpdateDHCPOptOnPort

ironic.dhcp.neutron module

class ironic.dhcp.neutron.**NeutronDHCPApi**

Bases: *ironic.dhcp.base.BaseDHCP*

API for communicating to neutron 2.x API.

get_ip_addresses(*task*)

Get IP addresses for all ports/portgroups in *task*.

Parameters **task** a TaskManager instance.

Returns List of IP addresses associated with tasks ports/portgroups.

update_dhcp_opts(*task*, *options*, *vifs=None*)

Send or update the DHCP BOOT options for this node.

Parameters

- **task** A TaskManager instance.

- **options** this will be a list of dicts, e.g.

```
[{'opt_name': '67',
  'opt_value': 'pxelinux.0',
  'ip_version': 4},
 {'opt_name': '66',
  'opt_value': '123.123.123.456',
  'ip_version': 4}]
```

- **vifs** a dict of Neutron port/portgroup dicts to update DHCP options on. The port/portgroup dict key should be Ironic port UUIDs, and the values should be Neutron port UUIDs, e.g.

```
{'ports': {'port.uuid': vif.id},
 'portgroups': {'portgroup.uuid': vif.id}}
If the value is None, will get the list of
ports/portgroups from the Ironic port/portgroup
objects.
```

update_port_dhcp_opts(*port_id*, *dhcp_options*, *token=None*, *context=None*)

Update a ports attributes.

Update one or more DHCP options on the specified port. For the relevant API spec, see https://docs.openstack.org/api-ref/network/v2/index.html#update-port

Parameters

- **port_id** designate which port these attributes will be applied to.
- **dhcp_options** this will be a list of dicts, e.g.

```
[{'opt_name': '67',
  'opt_value': 'pxelinux.0',
  'ip_version': 4},
 {'opt_name': '66',
  'opt_value': '123.123.123.456'},
  'ip_version': 4}]
```

- **token** optional auth token. Deprecated, use context.
- **context** (ironic.common.context.RequestContext) request context

Raises FailedToUpdateDHCPOptOnPort

ironic.dhcp.none module

class ironic.dhcp.none.**NoneDHCPApi**

Bases: *ironic.dhcp.base.BaseDHCP*

No-op DHCP API.

get_ip_addresses(*task*)

Get IP addresses for all ports/portgroups in *task*.

Parameters **task** A TaskManager instance.

Returns List of IP addresses associated with tasks ports and portgroups.

update_dhcp_opts(*task*, *options*, *vifs=None*)
Send or update the DHCP BOOT options for this node.

Parameters

- **task** A TaskManager instance.
- **options** this will be a list of dicts, e.g.

```
[{'opt_name': '67',
  'opt_value': 'pxelinux.0',
  'ip_version': 4},
 {'opt_name': '66',
  'opt_value': '123.123.123.456',
  'ip_version': 4}]
```

- **vifs** A dict with keys ports and portgroups and dicts as values. Each dict has key/value pairs of the form <ironic UUID>:<neutron port UUID>. e.g.

```
{'ports': {'port.uuid': vif.id},
 'portgroups': {'portgroup.uuid': vif.id}}
```

  If the value is None, will get the list of ports/portgroups from the Ironic port/portgroup objects.

Raises FailedToUpdateDHCPOptOnPort

update_port_dhcp_opts(*port_id*, *dhcp_options*, *token=None*, *context=None*)
Update one or more DHCP options on the specified port.

Parameters

- **port_id** designate which port these attributes will be applied to.
- **dhcp_options** this will be a list of dicts, e.g.

```
[{'opt_name': '67',
  'opt_value': 'pxelinux.0',
  'ip_version': 4},
 {'opt_name': '66',
  'opt_value': '123.123.123.456',
  'ip_version': 4}]
```

- **token** An optional authentication token. Deprecated, use context
- **context** (ironic.common.context.RequestContext) request context

Raises FailedToUpdateDHCPOptOnPort

Module contents

ironic.drivers package

Subpackages

ironic.drivers.modules package

Subpackages

ironic.drivers.modules.ansible package

Submodules

ironic.drivers.modules.ansible.deploy module

Ansible deploy interface

class ironic.drivers.modules.ansible.deploy.**AnsibleDeploy**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.agent_base.HeartbeatMixin*, *ironic.drivers.modules.agent_base.AgentOobStepsMixin*, *ironic.drivers.base.DeployInterface*

Interface for deploy-related actions.

clean_up(*task*)

Clean up the deployment environment for this node.

collect_deploy_logs = False

deploy(*task*)

Perform a deployment to a node.

execute_clean_step(*task*, *step*)

Execute a clean step.

Parameters

- **task** a TaskManager object containing the node
- **step** a clean step dictionary to execute

Returns None

get_clean_steps(*task*)

Get the list of clean steps from the file.

Parameters **task** a TaskManager object containing the node

Returns A list of clean step dictionaries

get_properties()

Return the properties of the interface.

in_core_deploy_step(*task*)

Check if we are in the deploy.deploy deploy step.

Assumes that we are in the DEPLOYWAIT state.

Parameters **task** a TaskManager instance

Returns True if the current deploy step is deploy.deploy.

prepare(*task*)
Prepare the deployment environment for this node.

prepare_cleaning(*task*)
Boot into the ramdisk to prepare for cleaning.

Parameters **task** a TaskManager object containing the node

Raises *NodeCleaningFailure* if the previous cleaning ports cannot be removed or if new cleaning ports cannot be created

Returns None or states.CLEANWAIT for async prepare.

process_next_step(*task*, *step_type*)
Start the next clean/deploy step if the previous one is complete.

Parameters

- **task** a TaskManager instance
- **step_type** clean or deploy

take_over(*task*)
Take over management of this tasks node from a dead conductor.

If conductors hosts maintain a static relationship to nodes, this method should be implemented by the driver to allow conductors to perform the necessary work during the remapping of nodes to conductors when a conductor joins or leaves the cluster.

For example, the PXE driver has an external dependency: Neutron must forward DHCP BOOT requests to a conductor which has prepared the tftpboot environment for the given node. When a conductor goes offline, another conductor must change this setting in Neutron as part of remapping that nodes control to itself. This is performed within the *takeover* method.

Parameters **task** A TaskManager instance containing the node to act on.

tear_down(*task*)
Tear down a previous deployment on the tasks node.

tear_down_agent(*task*)
A deploy step to tear down the agent.

Shuts down the machine and removes it from the provisioning network.

Parameters **task** a TaskManager object containing the node

tear_down_cleaning(*task*)
Clean up the PXE and DHCP files after cleaning.

Parameters **task** a TaskManager object containing the node

Raises *NodeCleaningFailure* if the cleaning ports cannot be removed

validate(*task*)
Validate the driver-specific Node deployment info.

write_image(*task*)

exception ironic.drivers.modules.ansible.deploy.**PlaybookNotFound**(*message=None*, ***kwargs*)

Bases: ironic_lib.exception.IronicException

Module contents

ironic.drivers.modules.drac package

Submodules

ironic.drivers.modules.drac.bios module

DRAC BIOS configuration specific methods

class ironic.drivers.modules.drac.bios.**DracRedfishBIOS**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.redfish.bios.RedfishBIOS*

iDRAC Redfish interface for BIOS settings-related actions.

Presently, this class entirely defers to its base class, a generic, vendor-independent Redfish interface. Future resolution of Dell EMC- specific incompatibilities and introduction of vendor value added should be implemented by this class.

class ironic.drivers.modules.drac.bios.**DracWSManBIOS**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.BIOSInterface*

BIOSInterface Implementation for iDRAC.

apply_configuration(*task*, *settings*)

Apply the BIOS configuration to the node

Parameters

- **task** a TaskManager instance containing the node to act on
- **settings** List of BIOS settings to apply

Raises DRACOperationError upon an error from python-dracclient

Returns states.CLEANWAIT (cleaning) or states.DEPLOYWAIT (deployment) if configuration is in progress asynchronously or None if it is completed.

cache_bios_settings(*task*)

Store or update the current BIOS settings for the node.

Get the current BIOS settings and store them in the bios_settings database table.

Parameters **task** a TaskManager instance containing the node to act on.

Raises DracOperationError on an error from python-dracclient

factory_reset(*task*)

Reset the BIOS settings of the node to the factory default.

This uses the Lifecycle Controller configuration to perform BIOS configuration reset. Leveraging the python-dracclient methods already available.

Parameters **task** a TaskManager instance containing the node to act on

Raises DracOperationError on an error from python-dracclient

Returns states.CLEANWAIT (cleaning) or states.DEPLOYWAIT (deployment) if reset is in progress asynchronously or None if it is completed.

get_properties()
Return the properties of the BIOS Interface

Returns dictionary of <property name>: <property description> entries

validate(*task*)
Validates the driver-specific information used by the idrac BMC

Parameters **task** a TaskManager instance containing the node to act on

Raises InvalidParameterValue if some mandatory information is missing on the node or on invalid inputs

ironic.drivers.modules.drac.bios.**abandon_config**(*task*)
Abandons uncommitted changes added by set_config

Parameters **task** a TaskManager instance containing the node to act on.

Raises DracOperationError on an error from python-dracclient.

ironic.drivers.modules.drac.bios.**commit_config**(*task*, *reboot=False*)
Commits pending changes added by set_config

Parameters

- **task** a TaskManager instance containing the node to act on.
- **reboot** indicates whether a reboot job should be automatically created with the config job.

Raises DracOperationError on an error from python-dracclient.

Returns the job_id key with the id of the newly created config job.

ironic.drivers.modules.drac.bios.**get_config**(*node*)
Get the BIOS configuration.

The BIOS settings look like:

```
{'EnumAttrib': {'name': 'EnumAttrib',
                'current_value': 'Value',
                'pending_value': 'New Value', # could also be None
                'read_only': False,
                'possible_values': ['Value', 'New Value', 'None']},
 'StringAttrib': {'name': 'StringAttrib',
                  'current_value': 'Information',
                  'pending_value': None,
                  'read_only': False,
                  'min_length': 0,
                  'max_length': 255,
                  'pcre_regex': '^[0-9A-Za-z]{0,255}$'},
 'IntegerAttrib': {'name': 'IntegerAttrib',
                   'current_value': 0,
                   'pending_value': None,
```

(continues on next page)

(continued from previous page)

```
                    'read_only': True,
                    'lower_bound': 0,
                    'upper_bound': 65535}}
```

Parameters **node** an ironic node object.

Raises DracOperationError on an error from python-dracclient.

Returns a dictionary containing BIOS settings

The above values are only examples, of course. BIOS attributes exposed via this API will always be either an enumerated attribute, a string attribute, or an integer attribute. All attributes have the following parameters:

Parameters

- **name** is the name of the BIOS attribute.
- **current_value** is the current value of the attribute. It will always be either an integer or a string.
- **pending_value** is the new value that we want the attribute to have. None means that there is no pending value.
- **read_only** indicates whether this attribute can be changed. Trying to change a read-only value will result in an error. The read-only flag can change depending on other attributes. A future version of this call may expose the dependencies that indicate when that may happen.

Enumerable attributes also have the following parameters:

Parameters **possible_values** is an array of values it is permissible to set the attribute to.

String attributes also have the following parameters:

Parameters

- **min_length** is the minimum length of the string.
- **max_length** is the maximum length of the string.
- **pcre_regex** is a PCRE compatible regular expression that the string must match. It may be None if the string is read only or if the string does not have to match any particular regular expression.

Integer attributes also have the following parameters:

Parameters

- **lower_bound** is the minimum value the attribute can have.
- **upper_bound** is the maximum value the attribute can have.

ironic.drivers.modules.drac.bios.**set_config**(*task*, ***kwargs*)

Sets the pending_value parameter for each of the values passed in.

Parameters

- **task** a TaskManager instance containing the node to act on.

- **kwargs** a dictionary of {AttributeName: NewValue}

Raises DracOperationError on an error from python-dracclient.

Returns A dictionary containing the is_commit_required key with a boolean value indicating whether commit_config() needs to be called to make the changes, and the is_reboot_required key which has a value of true or false. This key is used to indicate to the commit_config() call if a reboot should be performed.

ironic.drivers.modules.drac.boot module

class ironic.drivers.modules.drac.boot.**DracRedfishVirtualMediaBoot**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.redfish.boot.RedfishVirtualMediaBoot*

iDRAC Redfish interface for virtual media boot-related actions.

Virtual Media allows booting the system from virtual CD/DVD drive containing user image that BMC inserts into the drive.

The CD/DVD images must be in ISO format and (depending on BMC implementation) could be pulled over HTTP, served as iSCSI targets or NFS volumes.

The baseline boot workflow is mostly based on the standard Redfish virtual media boot interface, which looks like this:

1. Pull kernel, ramdisk and ESP if UEFI boot is requested (FAT partition image with EFI boot loader) images
2. Create bootable ISO out of images (#1), push it to Glance and pass to the BMC as Swift temporary URL
3. Optionally create floppy image with desired system configuration data, push it to Glance and pass to the BMC as Swift temporary URL
4. Insert CD/DVD and (optionally) floppy images and set proper boot mode

For building deploy or rescue ISO, redfish boot interface uses *deploy_kernel*/*deploy_ramdisk* or *rescue_kernel*/*rescue_ramdisk* properties from *[instance_info]* or *[driver_info]*.

For building boot (user) ISO, redfish boot interface seeks *kernel_id* and *ramdisk_id* properties in the Glance image metadata found in *[instance_info]image_source* node property.

iDRAC virtual media boot interface only differs by the way how it sets the node to boot from a virtual media device - this is done via OEM action call implemented in Dell sushy OEM extension package.

VIRTUAL_MEDIA_DEVICES = {'cdrom': VirtualMediaType.CD, 'floppy': VirtualMediaType.FLOPPY}

ironic.drivers.modules.drac.common module

Common functionalities shared between different DRAC modules.

ironic.drivers.modules.drac.common.**get_drac_client**(*node*)
Returns a DRACClient object from python-dracclient library.

Parameters **node** an ironic node object.

Returns a DRACClient object.

Raises InvalidParameterValue if mandatory information is missing on the node or on invalid input.

ironic.drivers.modules.drac.common.**parse_driver_info**(*node*)
Parse a nodes driver_info values.

Parses the driver_info of the node, reads default values and returns a dict containing the combination of both.

Parameters **node** an ironic node object.

Returns a dict containing information from driver_info and default values.

Raises InvalidParameterValue if some mandatory information is missing on the node or on invalid inputs.

ironic.drivers.modules.drac.inspect module

DRAC inspection interface

class ironic.drivers.modules.drac.inspect.**DracInspect**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.drac.inspect.DracWSManInspect*

Class alias of class DracWSManInspect.

This class provides ongoing support of the deprecated idrac inspect interface implementation entrypoint.

All bug fixes and new features should be implemented in its base class, DracWSManInspect. That makes them available to both the deprecated idrac and new idrac-wsman entrypoints. Such changes should not be made to this class.

class ironic.drivers.modules.drac.inspect.**DracRedfishInspect**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.redfish.inspect.RedfishInspect*

iDRAC Redfish interface for inspection-related actions.

inspect_hardware(*task*)
Inspect hardware to get the hardware properties.

Inspects hardware to get the essential properties. It fails if any of the essential properties are not received from the node.

Parameters **task** a TaskManager instance.

Raises HardwareInspectionFailure if essential properties could not be retrieved successfully.

Returns The resulting state of inspection.

class ironic.drivers.modules.drac.inspect.**DracWSManInspect**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.InspectInterface*

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

inspect_hardware(*task*)
Inspect hardware.

Inspect hardware to obtain the essential & additional hardware properties.

Parameters **task** a TaskManager instance containing the node to act on.

Raises HardwareInspectionFailure, if unable to get essential hardware properties.

Returns states.MANAGEABLE

validate(*task*)
Validate the driver-specific info supplied.

This method validates whether the driver_info property of the supplied node contains the required information for this driver to manage the node.

Parameters **task** a TaskManager instance containing the node to act on.

Raises InvalidParameterValue if required driver_info attribute is missing or invalid on the node.

ironic.drivers.modules.drac.job module

DRAC Lifecycle job specific methods

ironic.drivers.modules.drac.job.**get_job**(*node*, *job_id*)
Get the details of a Lifecycle job of the node.

Parameters

- **node** an ironic node object.
- **job_id** ID of the Lifecycle job.

Returns a Job object from dracclient.

Raises DracOperationError on an error from python-dracclient.

ironic.drivers.modules.drac.job.**list_unfinished_jobs**(*node*)
List unfinished config jobs of the node.

Parameters **node** an ironic node object.

Returns a list of Job objects from dracclient.

Raises DracOperationError on an error from python-dracclient.

ironic.drivers.modules.drac.job.**validate_job_queue**(*node*, *name_prefix=None*)
Validates the job queue on the node.

It raises an exception if an unfinished configuration job exists. :param node: an ironic node object. :param name_prefix: A name prefix for jobs to validate. :raises: DracOperationError on an error from python-dracclient.

ironic.drivers.modules.drac.job.**wait_for_job_completion**(*node*, *retries=240*)
Wait for job to complete

It will wait for the job to complete for 20 minutes and raises timeout if job never complete within given interval of time. :param node: an ironic node object. :param retries: no of retries to make conductor wait. :raises: DracOperationError on exception raised from python-dracclient or a timeout while waiting for job completion.

ironic.drivers.modules.drac.management module

DRAC management interface

class ironic.drivers.modules.drac.management.**DracManagement**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.drac.management.DracWSManManagement*

Class alias of class DracWSManManagement.

This class provides ongoing support of the deprecated idrac management interface implementation entrypoint.

All bug fixes and new features should be implemented in its base class, DracWSManManagement. That makes them available to both the deprecated idrac and new idrac-wsman entrypoints. Such changes should not be made to this class.

class ironic.drivers.modules.drac.management.**DracRedfishManagement**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.redfish.management.RedfishManagement*

iDRAC Redfish interface for management-related actions.

EXPORT_CONFIGURATION_ARGSINFO = {'export_configuration_location': {'description': 'URL of location to save the configuration to.', 'required': True}}

IMPORT_CONFIGURATION_ARGSINFO = {'import_configuration_location': {'description': 'URL of location to fetch desired configuration from.', 'required': True}}

IMPORT_EXPORT_CONFIGURATION_ARGSINFO = {'export_configuration_location': {'description': 'URL of location to save the configuration to.', 'required': True}, 'import_configuration_location': {'description': 'URL of location to fetch desired configuration from.', 'required': True}}

clear_job_queue(*task*)
Clear iDRAC job queue.

Parameters **task** a TaskManager instance containing the node to act on.

Raises RedfishError on an error.

export_configuration(*task*, *export_configuration_location*)
Export the configuration of the server.

Exports the configuration of the server against which the step is run and stores it in specific format in indicated location.

Uses Dells Server Configuration Profile (SCP) from *sushy-oem-idrac* library to get ALL configuration for cloning.

Parameters

- **task** A task from TaskManager.
- **export_configuration_location** URL of location to save the configuration to.

Raises MissingParameterValue if missing configuration name of a file to save the configuration to

Raises DracOperatationError when no managagers for Redfish system found or configuration export from SCP failed

Raises RedfishError when loading OEM extension failed

import_configuration(*task*, *import_configuration_location*)

Import and apply the configuration to the server.

Gets pre-created configuration from storage by given location and imports that into given server. Uses Dells Server Configuration Profile (SCP).

Parameters

- **task** A task from TaskManager.
- **import_configuration_location** URL of location to fetch desired configuration from.

Raises MissingParameterValue if missing configuration name of a file to fetch the configuration from

import_export_configuration(*task*, *import_configuration_location*, *export_configuration_location*)

Import and export configuration in one go.

Gets pre-created configuration from storage by given name and imports that into given server. After that exports the configuration of the server against which the step is run and stores it in specific format in indicated storage as configured by Ironic.

Parameters

- **import_configuration_location** URL of location to fetch desired configuration from.
- **export_configuration_location** URL of location to save the configuration to.

known_good_state(*task*)

Reset iDRAC to known good state.

An iDRAC is reset to a known good state by resetting it and clearing its job queue.

Parameters **task** a TaskManager instance containing the node to act on.

Raises RedfishError on an error.

reset_idrac(*task*)

Reset the iDRAC.

Parameters **task** a TaskManager instance containing the node to act on.

Raises RedfishError on an error.

class ironic.drivers.modules.drac.management.**DracWSManManagement**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.ManagementInterface*

clear_job_queue(*task*)
Clear the job queue.

Parameters **task** a TaskManager instance containing the node to act on.

Returns None if it is completed.

Raises DracOperationError on an error from python-dracclient.

get_boot_device(*task*)
Get the current boot device for a node.

Returns the current boot device of the node.

Parameters **task** a TaskManager instance containing the node to act on.

Raises DracOperationError on an error from python-dracclient.

Returns

a dictionary containing:

boot_device the boot device, one of *ironic.common.boot_devices* or None if it is unknown.

persistent whether the boot device will persist to all future boots or not, None if it is unknown.

get_properties()
Return the properties of the interface.

get_sensors_data(*task*)
Get sensors data.

Parameters **task** a TaskManager instance.

Raises FailedToGetSensorData when getting the sensor data fails.

Raises FailedToParseSensorData when parsing sensor data fails.

Returns returns a consistent format dict of sensor data grouped by sensor type, which can be processed by Ceilometer.

get_supported_boot_devices(*task*)
Get a list of the supported boot devices.

Parameters **task** a TaskManager instance containing the node to act on.

Returns A list with the supported boot devices defined in *ironic.common.boot_devices*.

known_good_state(*task*)
Reset the iDRAC, Clear the job queue.

Parameters **task** a TaskManager instance containing the node to act on.

Returns None if it is completed.

Raises DracOperationError on an error from python-dracclient.

reset_idrac(*task*)
Reset the iDRAC.

Parameters **task** a TaskManager instance containing the node to act on.

Returns None if it is completed.

Raises DracOperationError on an error from python-dracclient.

set_boot_device(*task*, *device*, *persistent=False*)
Set the boot device for a node.

Set the boot device to use on next reboot of the node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **device** the boot device, one of *ironic.common.boot_devices*.
- **persistent** Boolean value. True if the boot device will persist to all future boots, False if not. Default: False.

Raises InvalidParameterValue if an invalid boot device is specified.

validate(*task*)
Validate the driver-specific info supplied.

This method validates whether the driver_info property of the supplied node contains the required information for this driver to manage the node.

Parameters **task** a TaskManager instance containing the node to act on.

Raises InvalidParameterValue if required driver_info attribute is missing or invalid on the node.

ironic.drivers.modules.drac.management.**set_boot_device**(*node*, *device*, *persistent=False*)
Set the boot device for a node.

Set the boot device to use on next boot of the node.

Parameters

- **node** an ironic node object.
- **device** the boot device, one of *ironic.common.boot_devices*.
- **persistent** Boolean value. True if the boot device will persist to all future boots, False if not. Default: False.

Raises DracOperationError on an error from python-dracclient.

ironic.drivers.modules.drac.power module

DRAC power interface

class ironic.drivers.modules.drac.power.**DracPower**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.drac.power.DracWSManPower*

Class alias of class DracWSManPower.

This class provides ongoing support of the deprecated idrac power interface implementation entrypoint.

All bug fixes and new features should be implemented in its base class, DracWSManPower. That makes them available to both the deprecated idrac and new idrac-wsman entrypoints. Such changes should not be made to this class.

class ironic.drivers.modules.drac.power.**DracRedfishPower**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.redfish.power.RedfishPower*

iDRAC Redfish interface for power-related actions.

Presently, this class entirely defers to its base class, a generic, vendor-independent Redfish interface. Future resolution of Dell EMC- specific incompatibilities and introduction of vendor value added should be implemented by this class.

class ironic.drivers.modules.drac.power.**DracWSManPower**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.PowerInterface*

Interface for power-related actions.

get_power_state(*task*)
Return the power state of the node.

Parameters **task** a TaskManager instance containing the node to act on.

Returns the power state, one of *ironic.common.states*.

Raises InvalidParameterValue if required DRAC credentials are missing.

Raises DracOperationError on an error from python-dracclient.

get_properties()
Return the properties of the interface.

reboot(*task*, *timeout=None*)
Perform a reboot of the tasks node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **timeout** timeout (in seconds). Unsupported by this interface.

Raises InvalidParameterValue if required DRAC credentials are missing.

Raises DracOperationError on an error from python-dracclient.

set_power_state(*task*, *power_state*, *timeout=None*)
Set the power state of the node.

Parameters

- **task** a TaskManager instance containing the node to act on.

- **power_state** a power state from *ironic.common.states*.
- **timeout** Time to wait for the node to reach the requested state. When requested state is reboot, not used as not waiting then.

Raises InvalidParameterValue if required DRAC credentials are missing.

Raises DracOperationError on an error from python-dracclient.

validate(*task*)

Validate the driver-specific Node power info.

This method validates whether the driver_info property of the supplied node contains the required information for this driver to manage the power state of the node.

Parameters **task** a TaskManager instance containing the node to act on.

Raises InvalidParameterValue if required driver_info attribute is missing or invalid on the node.

ironic.drivers.modules.drac.raid module

DRAC RAID specific methods

class ironic.drivers.modules.drac.raid.**DracRAID**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.drac.raid.DracWSManRAID*

Class alias of class DracWSManRAID.

This class provides ongoing support of the deprecated idrac RAID interface implementation entrypoint.

All bug fixes and new features should be implemented in its base class, DracWSManRAID. That makes them available to both the deprecated idrac and new idrac-wsman entrypoints. Such changes should not be made to this class.

class ironic.drivers.modules.drac.raid.**DracRedfishRAID**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.redfish.raid.RedfishRAID*

iDRAC Redfish interface for RAID related actions.

Includes iDRAC specific adjustments for RAID related actions.

create_configuration(*task*, *create_root_volume=True*, *create_nonroot_volumes=True*, *delete_existing=False*)

Create RAID configuration on the node.

This method creates the RAID configuration as read from node.target_raid_config. This method by default will create all logical disks.

Parameters

- **task** TaskManager object containing the node.
- **create_root_volume** Setting this to False indicates not to create root volume that is specified in the nodes target_raid_config. Default value is True.
- **create_nonroot_volumes** Setting this to False indicates not to create non-root volumes (all except the root volume) in the nodes target_raid_config. Default value is True.

- **delete_existing** Setting this to True indicates to delete RAID configuration prior to creating the new configuration. Default is False.

Returns states.CLEANWAIT if RAID configuration is in progress asynchronously or None if it is complete.

Raises RedfishError if there is an error creating the configuration

delete_configuration(*task*)
Delete RAID configuration on the node.

Parameters **task** TaskManager object containing the node.

Returns states.CLEANWAIT (cleaning) or states.DEPLOYWAIT (deployment) if deletion is in progress asynchronously or None if it is complete.

post_delete_configuration(*task*, *raid_configs*, *return_state=None*)
Perform post delete_configuration action to commit the config.

Clears foreign configuration for all RAID controllers.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **raid_configs** a list of dictionaries containing the RAID configuration operation details.
- **return_state** state to return based on operation being invoked

pre_create_configuration(*task*, *logical_disks_to_create*)
Perform required actions before creating config.

Converts any physical disks of selected controllers to RAID mode if in non-RAID mode.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **logical_disks_to_create** list of logical disks to create.

Returns updated list of logical disks to create

class ironic.drivers.modules.drac.raid.**DracWSManRAID**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.RAIDInterface*

apply_configuration(*task*, *raid_config*, *create_root_volume=True*, *create_nonroot_volumes=False*, *delete_existing=True*)
Applies RAID configuration on the given node.

Parameters

- **task** A TaskManager instance.
- **raid_config** The RAID configuration to apply.
- **create_root_volume** Setting this to False indicates not to create root volume that is specified in raid_config. Default value is True.
- **create_nonroot_volumes** Setting this to False indicates not to create non-root volumes (all except the root volume) in raid_config. Default value is True.

- **delete_existing** Setting this to True indicates to delete RAID configuration prior to creating the new configuration.

Raises InvalidParameterValue, if the RAID configuration is invalid.

Returns states.DEPLOYWAIT if RAID configuration is in progress asynchronously or None if it is complete.

create_configuration(*task*, *create_root_volume=True*, *create_nonroot_volumes=True*, *delete_existing=False*)

Create the RAID configuration.

This method creates the RAID configuration on the given node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **create_root_volume** If True, a root volume is created during RAID configuration. Otherwise, no root volume is created. Default is True.
- **create_nonroot_volumes** If True, non-root volumes are created. If False, no non-root volumes are created. Default is True.
- **delete_existing** Setting this to True indicates to delete RAID configuration prior to creating the new configuration. Default is False.

Returns states.CLEANWAIT (cleaning) or states.DEPLOYWAIT (deployment) if creation is in progress asynchronously or None if it is completed.

Raises MissingParameterValue, if node.target_raid_config is missing or empty.

Raises DracOperationError on an error from python-dracclient.

delete_configuration(*task*)

Delete the RAID configuration.

Parameters **task** a TaskManager instance containing the node to act on.

Returns states.CLEANWAIT (cleaning) or states.DEPLOYWAIT (deployment) if deletion is in progress asynchronously or None if it is completed.

Raises DracOperationError on an error from python-dracclient.

get_logical_disks(*task*)

Get the RAID configuration of the node.

Parameters **task** a TaskManager instance containing the node to act on.

Returns A dictionary of properties.

Raises DracOperationError on an error from python-dracclient.

get_properties()

Return the properties of the interface.

ironic.drivers.modules.drac.raid.**abandon_config**(*node*, *raid_controller*)

Deletes all pending changes on a RAID controller.

Parameters

- **node** an ironic node object.
- **raid_controller** id of the RAID controller.

Raises DracOperationError on an error from python-dracclient.

ironic.drivers.modules.drac.raid.**change_physical_disk_state**(*node*, *mode=None*, *controllers_to_physical_disk_ids=None*)

Convert disks RAID status

This method converts the requested physical disks from RAID to JBOD or vice versa. It does this by only converting the disks that are not already in the correct state.

Parameters

- **node** an ironic node object.
- **mode** the mode to change the disks either to RAID or JBOD.
- **controllers_to_physical_disk_ids** Dictionary of controllers and corresponding disk ids to convert to the requested mode.

Returns a dictionary containing: - conversion_results, a dictionary that maps controller ids to the conversion results for that controller. The conversion results are a dict that contains: - The is_commit_required key with the value always set to True indicating that a config job must be created to complete disk conversion. - The is_reboot_required key with a RebootRequired enumerated value indicating whether the server must be rebooted to complete disk conversion.

Raises DRACOperationError on an error from python-dracclient.

ironic.drivers.modules.drac.raid.**clear_foreign_config**(*node*, *raid_controller*)

Free up the foreign drives.

Parameters

- **node** an ironic node object.
- **raid_controller** id of the RAID controller.

Returns a dictionary containing - The is_commit_required needed key with a boolean value indicating whether a config job must be created for the values to be applied. - The is_reboot_required key with a RebootRequired enumerated value indicating whether the server must be rebooted to clear foreign configuration.

Raises DracOperationError on an error from python-dracclient.

ironic.drivers.modules.drac.raid.**commit_config**(*node*, *raid_controller*, *reboot=False*, *realtime=False*)

Apply all pending changes on a RAID controller.

Parameters

- **node** an ironic node object.
- **raid_controller** id of the RAID controller.
- **reboot** indicates whether a reboot job should be automatically created with the config job. (optional, defaults to False)
- **realtime** indicates RAID controller supports realtime. (optional, defaults to False)

Returns id of the created job

Raises DracOperationError on an error from python-dracclient.

ironic.drivers.modules.drac.raid.**create_virtual_disk**(*node*, *raid_controller*, *physical_disks*, *raid_level*, *size_mb*, *disk_name=None*, *span_length=None*, *span_depth=None*)

Create a single virtual disk on a RAID controller.

The created virtual disk will be in pending state. The DRAC card will do the actual configuration once the changes are applied by calling the `commit_config` method.

Parameters

- **node** an ironic node object.
- **raid_controller** id of the RAID controller.
- **physical_disks** ids of the physical disks.
- **raid_level** RAID level of the virtual disk.
- **size_mb** size of the virtual disk.
- **disk_name** name of the virtual disk. (optional)
- **span_depth** Number of spans in virtual disk. (optional)
- **span_length** Number of disks per span. (optional)

Returns a dictionary containing the commit_needed key with a boolean value indicating whether a config job must be created for the values to be applied.

Raises DracOperationError on an error from python-dracclient.

ironic.drivers.modules.drac.raid.**delete_virtual_disk**(*node*, *virtual_disk*)

Delete a single virtual disk on a RAID controller.

The deleted virtual disk will be in pending state. The DRAC card will do the actual configuration once the changes are applied by calling the `commit_config` method.

Parameters

- **node** an ironic node object.
- **virtual_disk** id of the virtual disk.

Returns a dictionary containing the commit_needed key with a boolean value indicating whether a config job must be created for the values to be applied.

Raises DracOperationError on an error from python-dracclient.

ironic.drivers.modules.drac.raid.**list_physical_disks**(*node*)

List the physical disks of the node.

Parameters **node** an ironic node object.

Returns a list of PhysicalDisk objects from dracclient.

Raises DracOperationError on an error from python-dracclient.

ironic.drivers.modules.drac.raid.**list_raid_controllers**(*node*)

List the RAID controllers of the node.

Parameters **node** an ironic node object.

Returns a list of RAIDController objects from dracclient.

Raises DracOperationError on an error from python-dracclient.

ironic.drivers.modules.drac.raid.**list_raid_settings**(*node*)
List the RAID configuration settings

Parameters **node** an ironic node object.

Returns a dictionary with the RAID settings using InstanceID as the key. The attributes are RAIDEnumerableAttribute, RAIDStringAttribute and RAIDIntegerAttribute objects.

Raises DRACOperationFailed on error reported back by the DRAC interface

ironic.drivers.modules.drac.raid.**list_virtual_disks**(*node*)
List the virtual disks of the node.

Parameters **node** an ironic node object.

Returns a list of VirtualDisk objects from dracclient.

Raises DracOperationError on an error from python-dracclient.

ironic.drivers.modules.drac.raid.**set_raid_settings**(*node*, *controller_fqdd*, *settings*)
Sets the RAID configuration

It sets the pending_value parameter for each of the attributes passed in. For the values to be applied, a config job must be created.

Parameters

- **node** an ironic node object.
- **controller_fqdd** the ID of the RAID controller.
- **settings** a dictionary containing the proposed values, with each key being the name of attribute and the value being the proposed value.

Returns a dictionary containing: - The is_commit_required key with a boolean value indicating whether a config job must be created for the values to be applied. - The is_reboot_required key with a RebootRequired enumerated value indicating whether the server must be rebooted for the values to be applied. Possible values are true and false.

Raises DRACOperationFailed on error reported back by the DRAC interface

ironic.drivers.modules.drac.utils module

ironic.drivers.modules.drac.utils.**execute_oem_manager_method**(*task*, *process_name*, *lambda_oem_func*)
Loads OEM manager and executes passed method on it.

Known iDRAC Redfish systems has only one manager, but as Redfish schema allows a list this method iterates through all values in case this changes in future. If there are several managers, this will try starting from the first in the list until the first success.

Parameters

- **task** a TaskManager instance.

- **process_name** user friendly name of method to be executed. Used in exception and log messages.
- **lambda_oem_func** method to execute as lambda function with input parameter OEM extension manager. Example: lambda m: m.reset_idrac()

Returns Returned value of lambda_oem_func

Raises RedfishError if cant execute OEM function either because there are no managers to the system, failed to load OEM extension or execution of the OEM method failed itself.

ironic.drivers.modules.drac.vendor_passthru module

DRAC vendor-passthru interface

class ironic.drivers.modules.drac.vendor_passthru.**DracRedfishVendorPassthru**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.redfish.vendor.RedfishVendorPassthru*

iDRAC Redfish interface for vendor_passthru.

Use the Redfish implementation for vendor passthru.

class ironic.drivers.modules.drac.vendor_passthru.**DracVendorPassthru**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.drac.vendor_passthru.DracWSManVendorPassthru*

Class alias of class DracWSManVendorPassthru.

This class provides ongoing support of the deprecated idrac vendor passthru interface implementation entrypoint.

All bug fixes and new features should be implemented in its base class, DracWSManVendorPassthru. That makes them available to both the deprecated idrac and new idrac-wsman entrypoints. Such changes should not be made to this class.

class ironic.drivers.modules.drac.vendor_passthru.**DracWSManVendorPassthru**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.VendorInterface*

Interface for DRAC specific methods.

abandon_bios_config(*task*, ***kwargs*)

Abandon a BIOS configuration job.

This method is used to abandon a BIOS configuration previously submitted through set_bios_config().

Parameters

- **task** a TaskManager instance containing the node to act on.
- **kwargs** not used.

Raises DracOperationError on an error from python-dracclient.

commit_bios_config(*task*, *reboot=False*, ***kwargs*)

Commit a BIOS configuration job.

This method is used to commit a BIOS configuration job. submitted through set_bios_config().

Parameters

- **task** a TaskManager instance containing the node to act on.
- **reboot** indicates whether a reboot job should be automatically created with the config job.
- **kwargs** not used.

Raises DracOperationError on an error from python-dracclient.

Returns A dictionary containing the `job_id` key with the id of the newly created config job, and the `reboot_required` key indicating whether the node needs to be rebooted to start the config job.

`get_bios_config`(*task*, ***kwargs*)
Get the BIOS configuration.

This method is used to retrieve the BIOS settings from a node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **kwargs** not used.

Raises DracOperationError on an error from python-dracclient.

Returns a dictionary containing BIOS settings.

`get_properties`()
Return the properties of the interface.

`list_unfinished_jobs`(*task*, ***kwargs*)
List unfinished config jobs of the node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **kwargs** not used.

Returns a dictionary containing the `unfinished_jobs` key; this key points to a list of dicts, with each dict representing a Job object.

Raises DracOperationError on an error from python-dracclient.

`set_bios_config`(*task*, ***kwargs*)
Change BIOS settings.

This method is used to change the BIOS settings on a node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **kwargs** a dictionary of {AttributeName: NewValue}

Raises DracOperationError on an error from python-dracclient.

Returns A dictionary containing the `is_commit_required` key with a Boolean value indicating whether commit_bios_config() needs to be called to make the changes, and the `is_reboot_required` key with a value of true or false. This key is used to indicate to the commit_bios_config() call if a reboot should be performed.

`validate`(*task*, ***kwargs*)

Validate the driver-specific info supplied.

This method validates whether the driver_info property of the supplied node contains the required information for this driver to manage the power state of the node.

Parameters

- **`task`** a TaskManager instance containing the node to act on.
- **`kwargs`** not used.

Raises InvalidParameterValue if required driver_info attribute is missing or invalid on the node.

Module contents

ironic.drivers.modules.ibmc package

Submodules

ironic.drivers.modules.ibmc.management module

iBMC Management Interface

`class ironic.drivers.modules.ibmc.management.IBMCManagement`(**args*, ***kwargs*)

Bases: *ironic.drivers.base.ManagementInterface*

`get_boot_device`(*task*)

Get the current boot device for a node.

Parameters **`task`** A task from TaskManager.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

Raises IBMCConnectionError when it fails to connect to iBMC

Raises IBMCError when iBMC responses an error information

Returns

a dictionary containing:

boot_device the boot device, one of *ironic.common.boot_devices* or None if it is unknown.

persistent Boolean value or None, True if the boot device persists, False otherwise. None if its disabled.

get_boot_mode(*task*)
Get the current boot mode for a node.

Provides the current boot mode of the node.

Parameters **task** A task from TaskManager.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

Raises IBMCConnectionError when it fails to connect to iBMC

Raises IBMCError when iBMC responses an error information

Returns The boot mode, one of `ironic.common.boot_mode` or None if it is unknown.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

get_sensors_data(*task*)
Get sensors data.

Not implemented for this driver.

Raises NotImplementedError

get_supported_boot_devices(*task*)
Get a list of the supported boot devices.

Parameters **task** a task from TaskManager.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

Raises IBMCConnectionError when it fails to connect to iBMC

Raises IBMCError when iBMC responses an error information

Returns A list with the supported boot devices defined in *ironic.common.boot_devices*.

get_supported_boot_modes(*task*)
Get a list of the supported boot modes.

Parameters **task** A task from TaskManager.

Returns A list with the supported boot modes defined in *ironic.common.boot_modes*. If boot mode support cant be determined, empty list is returned.

inject_nmi(*task*)
Inject NMI, Non Maskable Interrupt.

Inject NMI (Non Maskable Interrupt) for a node immediately.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

Raises IBMCConnectionError when it fails to connect to iBMC

Raises IBMCError when iBMC responses an error information

`set_boot_device`(*task*, *device*, *persistent=False*)
Set the boot device for a node.

Parameters

- **task** A task from TaskManager.
- **device** The boot device, one of `ironic.common.boot_device`.
- **persistent** Boolean value. True if the boot device will persist to all future boots, False if not. Default: False.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

Raises IBMCConnectionError when it fails to connect to iBMC

Raises IBMCError when iBMC responses an error information

`set_boot_mode`(*task*, *mode*)
Set the boot mode for a node.

Set the boot mode to use on next reboot of the node.

Parameters

- **task** A task from TaskManager.
- **mode** The boot mode, one of *ironic.common.boot_modes*.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

Raises IBMCConnectionError when it fails to connect to iBMC

Raises IBMCError when iBMC responses an error information

`validate`(*task*)
Validates the driver information needed by the iBMC driver.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

ironic.drivers.modules.ibmc.mappings module

iBMC and Ironic constants mapping

ironic.drivers.modules.ibmc.power module

iBMC Power Interface

class ironic.drivers.modules.ibmc.power.**IBMCPower**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.PowerInterface*

get_power_state(*task*)

Get the current power state of the tasks node.

Parameters **task** A TaskManager instance containing the node to act on.

Returns A power state. One of *ironic.common.states*.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

Raises IBMCConnectionError when it fails to connect to iBMC

Raises IBMCError when iBMC responses an error information

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

get_supported_power_states(*task*)

Get a list of the supported power states.

Parameters **task** A TaskManager instance containing the node to act on. Not used by this driver at the moment.

Returns A list with the supported power states defined in *ironic.common.states*.

reboot(*task*, *timeout=None*)

Perform a hard reboot of the tasks node.

Parameters

- **task** A TaskManager instance containing the node to act on.
- **timeout** Time to wait for the node to become powered on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue if a required parameter is missing.

Raises IBMCConnectionError when it fails to connect to iBMC

Raises IBMCError when iBMC responses an error information

set_power_state(*task*, *power_state*, *timeout=None*)

Set the power state of the tasks node.

Parameters

- **task** A TaskManager instance containing the node to act on.
- **power_state** Any power state from *ironic.common.states*.
- **timeout** Time to wait for the node to reach the requested state.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue if a required parameter is missing.

Raises IBMCConnectionError when it fails to connect to iBMC

Raises IBMCError when iBMC responses an error information

`validate`(*task*)

Validates the driver information needed by the iBMC driver.

Parameters `task` A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

ironic.drivers.modules.ibmc.raid module

iBMC RAID configuration specific methods

`class` ironic.drivers.modules.ibmc.raid.`IbmcRAID`(**args*, ***kwargs*)

Bases: *ironic.drivers.base.RAIDInterface*

Implementation of RAIDInterface for iBMC.

`RAID_APPLY_CONFIGURATION_ARGSINFO = {'create_nonroot_volumes': {'description': "Setting this to 'False' indicates not to create non-root volumes (all except the root volume) in 'raid_config'. Default value is 'True'.", 'required': False}, 'create_root_volume': {'description': "Setting this to 'False' indicates not to create root volume that is specified in 'raid_config'. Default value is 'True'.", 'required': False}, 'delete_existing': {'description': "Setting this to 'True' indicates to delete existing RAID configuration prior to creating the new configuration. Default value is 'True'.", 'required': False}, 'raid_config': {'description': 'The RAID configuration to apply.', 'required': True}}`

`apply_configuration`(*task*, *raid_config*, *create_root_volume=True*, *create_nonroot_volumes=False*)

Applies RAID configuration on the given node.

Parameters

- `task` A TaskManager instance.
- `raid_config` The RAID configuration to apply.
- `create_root_volume` Setting this to False indicates not to create root volume that is specified in raid_config. Default value is True.
- `create_nonroot_volumes` Setting this to False indicates not to create non-root volumes (all except the root volume) in raid_config. Default value is True.
- `delete_existing` Setting this to True indicates to delete RAID configuration prior to creating the new configuration.

Raises InvalidParameterValue, if the RAID configuration is invalid.

Returns states.DEPLOYWAIT if RAID configuration is in progress asynchronously or None if it is complete.

create_configuration(*task*, *create_root_volume=True*, *create_nonroot_volumes=True*, *delete_existing=False*)

Create a RAID configuration.

This method creates a RAID configuration on the given node.

Parameters

- **task** a TaskManager instance.
- **create_root_volume** If True, a root volume is created during RAID configuration. Otherwise, no root volume is created. Default is True.
- **create_nonroot_volumes** If True, non-root volumes are created. If False, no non-root volumes are created. Default is True.
- **delete_existing** Setting this to True indicates to delete RAID configuration prior to creating the new configuration. Default is False.

Raises MissingParameterValue, if node.target_raid_config is missing or empty after skipping root volume and/or non-root volumes.

Raises IBMCError, on failure to execute step.

delete_configuration(*task*)

Delete the RAID configuration.

Parameters **task** a TaskManager instance containing the node to act on.

Returns states.CLEANWAIT if cleaning operation in progress asynchronously or states.DEPLOYWAIT if deploy operation in progress synchronously or None if it is completed.

Raises IBMCError, on failure to execute step.

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

ironic.drivers.modules.ibmc.utils module

iBMC Driver common utils

ironic.drivers.modules.ibmc.utils.**handle_ibmc_exception**(*action*)

Decorator to handle iBMC client exception.

Decorated functions must take a `TaskManager` as the first parameter.

ironic.drivers.modules.ibmc.utils.**parse_driver_info**(*node*)

Parse the information required for Ironic to connect to iBMC.

Parameters **node** an Ironic node object

Returns dictionary of parameters

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

ironic.drivers.modules.ibmc.utils.**revert_dictionary**(*d*)

ironic.drivers.modules.ibmc.vendor module

iBMC Vendor Interface

class ironic.drivers.modules.ibmc.vendor.**IBMCVendor**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.VendorInterface*

boot_up_seq(*task*, ***kwargs*)

List boot type order of the node.

Parameters

- **task** A TaskManager instance containing the node to act on.
- **kwargs** Not used.

Raises InvalidParameterValue if kwargs does not contain method.

Raises MissingParameterValue

Raises IBMCConnectionError when it fails to connect to iBMC

Raises IBMCError when iBMC responses an error information

Returns A dictionary, containing node boot up sequence, in ascending order.

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

get_raid_controller_list(*task*, ***kwargs*)

List RAID controllers summary info of the node.

Parameters

- **task** A TaskManager instance containing the node to act on.
- **kwargs** Not used.

Raises IBMCConnectionError when it fails to connect to iBMC

Raises IBMCError when iBMC responses an error information

Returns A list of dictionaries, every dictionary represents a RAID controller summary of node.

validate(*task*, *method=None*, ***kwargs*)

Validate vendor-specific actions.

If invalid, raises an exception; otherwise returns None.

Parameters

- **task** A task from TaskManager.
- **method** Method to be validated
- **kwargs** Info for action.

Raises UnsupportedDriverExtension if method can not be mapped to the supported interfaces.

Raises InvalidParameterValue if kwargs does not contain method.

Raises MissingParameterValue

Module contents

ironic.drivers.modules.ilo package

Submodules

ironic.drivers.modules.ilo.bios module

iLO BIOS Interface

class ironic.drivers.modules.ilo.bios.**IloBIOS**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.BIOSInterface*

apply_configuration(*task*, *settings*)

Applies the provided configuration on the node.

Parameters

- **task** a TaskManager instance.
- **settings** Settings intended to be applied on the node.

Raises NodeCleaningFailure, on failure to execute of clean step.

Raises InstanceDeployFailure, on failure to execute of deploy step.

cache_bios_settings(*task*)

Store the BIOS settings in the database.

Parameters **task** a TaskManager instance.

Raises NodeCleaningFailure, on failure to execute of clean step.

Raises InstanceDeployFailure, on failure to execute of deploy step.

factory_reset(*task*)

Reset the BIOS settings to factory configuration.

Parameters **task** a TaskManager instance.

Raises NodeCleaningFailure, on failure to execute of clean step.

Raises InstanceDeployFailure, on failure to execute of deploy step.

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

validate(*task*)

Check that driver_info contains required ILO credentials.

Validates whether the driver_info property of the supplied tasks node contains the required credentials information.

Parameters **task** a task from TaskManager.

Raises InvalidParameterValue if required iLO parameters are not valid.

Raises MissingParameterValue if a required parameter is missing.

ironic.drivers.modules.ilo.boot module

Boot Interface for iLO drivers and its supporting methods.

class ironic.drivers.modules.ilo.boot.**IloPXEBoot**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.pxe.PXEBoot*

clean_up_instance(*task*)

Cleans up the boot of instance.

This method cleans up the PXE environment that was setup for booting the instance. It unlinks the instance kernel/ramdisk in the nodes directory in tftproot and removes its PXE config. In case of UEFI iSCSI booting, it cleans up iSCSI target information from the node.

Parameters **task** a task from TaskManager.

Returns None

Raises IloOperationError, if some operation on iLO failed.

prepare_instance(*task*)

Prepares the boot of instance.

This method prepares the boot of the instance after reading relevant information from the nodes instance_info. In case of netboot, it updates the dhcp entries and switches the PXE config. In case of localboot, it cleans up the PXE config. In case of boot from volume, it updates the iSCSI info onto iLO and sets the node to boot from UefiTarget boot device.

Parameters **task** a task from TaskManager.

Returns None

Raises IloOperationError, if some operation on iLO failed.

prepare_ramdisk(*task*, *ramdisk_params*)

Prepares the boot of Ironic ramdisk using PXE.

This method prepares the boot of the deploy or rescue ramdisk after reading relevant information from the nodes driver_info and instance_info.

Parameters

- **task** a task from TaskManager.
- **ramdisk_params** the parameters to be passed to the ramdisk.

Returns None

Raises MissingParameterValue, if some information is missing in nodes driver_info or instance_info.

Raises InvalidParameterValue, if some information provided is invalid.

Raises IronicException, if some power or set boot boot device operation failed on the node.

Raises IloOperationError, if some operation on iLO failed.

class ironic.drivers.modules.ilo.boot.**IloUefiHttpsBoot**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.BootInterface*

capabilities = ['ramdisk_boot']

clean_up_instance(*task*)

Cleans up the boot of instance.

This method cleans up the environment that was setup for booting the instance.

Parameters **task** A task from TaskManager.

Returns None

clean_up_ramdisk(*task*)

Cleans up the boot of ironic ramdisk.

This method cleans up the environment that was setup for booting the deploy ramdisk.

Parameters **task** A task from TaskManager.

Returns None

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

prepare_instance(*task*)

Prepares the boot of instance.

This method prepares the boot of the instance after reading relevant information from the nodes instance_info. It does the following depending on boot_option for deploy:

- If the boot_option requested for this deploy is local or image is a whole disk image, then it sets the node to boot from disk.
- Otherwise it finds/creates the boot ISO, sets the node boot option to UEFIHTTP and sets the URL as the boot ISO to boot the instance image.

Parameters **task** a task from TaskManager.

Returns None

Raises IloOperationError, if some operation on iLO failed.

Raises InstanceDeployFailure, if its try to boot iSCSI volume in BIOS boot mode.

prepare_ramdisk(*task*, *ramdisk_params*)

Prepares the boot of deploy ramdisk using UEFI-HTTPS boot.

This method prepares the boot of the deploy or rescue ramdisk after reading relevant information from the nodes driver_info and instance_info.

Parameters

- **task** a task from TaskManager.

- **ramdisk_params** the parameters to be passed to the ramdisk.

Returns None

Raises MissingParameterValue, if some information is missing in nodes driver_info or instance_info.

Raises InvalidParameterValue, if some information provided is invalid.

Raises IronicException, if some power or set boot boot device operation failed on the node.

Raises IloOperationError, if some operation on iLO failed.

validate(*task*)
Validate the deployment information for the tasks node.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

validate_inspection(*task*)
Validate that the node has required properties for inspection.

Parameters **task** A TaskManager instance with the node being checked

Raises MissingParameterValue if node is missing one or more required parameters

Raises UnsupportedDriverExtension

validate_rescue(*task*)
Validate that the node has required properties for rescue.

Parameters **task** a TaskManager instance with the node being checked

Raises MissingParameterValue if node is missing one or more required parameters

class ironic.drivers.modules.ilo.boot.**IloVirtualMediaBoot**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.BootInterface*

capabilities = ['iscsi_volume_boot', 'ramdisk_boot']

clean_up_instance(*task*)
Cleans up the boot of instance.

This method cleans up the environment that was setup for booting the instance. It ejects virtual media. In case of UEFI iSCSI booting, it cleans up iSCSI target information from the node.

Parameters **task** a task from TaskManager.

Returns None

Raises IloOperationError, if some operation on iLO failed.

clean_up_ramdisk(*task*)
Cleans up the boot of ironic ramdisk.

This method cleans up virtual media devices setup for the deploy or rescue ramdisk.

Parameters **task** a task from TaskManager.

Returns None

Raises IloOperationError, if some operation on iLO failed.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

prepare_instance(*task*)
Prepares the boot of instance.

This method prepares the boot of the instance after reading relevant information from the nodes instance_info. It does the following depending on boot_option for deploy:

- If the boot mode is uefi and its booting from volume, then it sets the iSCSI target info and node to boot from UefiTarget boot device.
- If not boot from volume and the boot_option requested for this deploy is local or image is a whole disk image, then it sets the node to boot from disk.
- Otherwise it finds/creates the boot ISO to boot the instance image, attaches the boot ISO to the bare metal and then sets the node to boot from CDROM.

Parameters **task** a task from TaskManager.

Returns None

Raises IloOperationError, if some operation on iLO failed.

Raises InstanceDeployFailure, if its try to boot iSCSI volume in BIOS boot mode.

prepare_ramdisk(*task*, *ramdisk_params*)
Prepares the boot of deploy ramdisk using virtual media.

This method prepares the boot of the deploy or rescue ramdisk after reading relevant information from the nodes driver_info and instance_info.

Parameters

- **task** a task from TaskManager.
- **ramdisk_params** the parameters to be passed to the ramdisk.

Returns None

Raises MissingParameterValue, if some information is missing in nodes driver_info or instance_info.

Raises InvalidParameterValue, if some information provided is invalid.

Raises IronicException, if some power or set boot boot device operation failed on the node.

Raises IloOperationError, if some operation on iLO failed.

validate(*task*)
Validate the deployment information for the tasks node.

Parameters **task** a TaskManager instance containing the node to act on.

Raises InvalidParameterValue, if some information is invalid.

Raises MissingParameterValue if kernel_id and ramdisk_id are missing in the Glance image or kernel and ramdisk not provided in instance_info for non-Glance image.

validate_inspection(*task*)
Validate that the node has required properties for inspection.

Parameters **task** A TaskManager instance with the node being checked

Raises MissingParameterValue if node is missing one or more required parameters

Raises UnsupportedDriverExtension

validate_rescue(*task*)
Validate that the node has required properties for rescue.

Parameters **task** a TaskManager instance with the node being checked

Raises MissingParameterValue if node is missing one or more required parameters

class ironic.drivers.modules.ilo.boot.**IloiPXEBoot**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.ipxe.iPXEBoot*

clean_up_instance(*task*)
Cleans up the boot of instance.

This method cleans up the PXE environment that was setup for booting the instance. It unlinks the instance kernel/ramdisk in the nodes directory in tftproot and removes its PXE config. In case of UEFI iSCSI booting, it cleans up iSCSI target information from the node.

Parameters **task** a task from TaskManager.

Returns None

Raises IloOperationError, if some operation on iLO failed.

prepare_instance(*task*)
Prepares the boot of instance.

This method prepares the boot of the instance after reading relevant information from the nodes instance_info. In case of netboot, it updates the dhcp entries and switches the PXE config. In case of localboot, it cleans up the PXE config. In case of boot from volume, it updates the iSCSI info onto iLO and sets the node to boot from UefiTarget boot device.

Parameters **task** a task from TaskManager.

Returns None

Raises IloOperationError, if some operation on iLO failed.

prepare_ramdisk(*task*, *ramdisk_params*)
Prepares the boot of Ironic ramdisk using PXE.

This method prepares the boot of the deploy or rescue ramdisk after reading relevant information from the nodes driver_info and instance_info.

Parameters

- **task** a task from TaskManager.
- **ramdisk_params** the parameters to be passed to the ramdisk.

Returns None

Raises MissingParameterValue, if some information is missing in nodes driver_info or instance_info.

Raises InvalidParameterValue, if some information provided is invalid.

Raises IronicException, if some power or set boot boot device operation failed on the node.

Raises IloOperationError, if some operation on iLO failed.

ironic.drivers.modules.ilo.boot.**parse_driver_info**(*node*, *mode='deploy'*)

Gets the driver specific Node deployment info.

This method validates whether the driver_info property of the supplied node contains the required information for this driver to deploy images to the node.

Parameters

- **node** a single Node.
- **mode** Label indicating a deploy or rescue operation being carried out on the node. Supported values are deploy and rescue. Defaults to deploy, indicating deploy operation is being carried out.

Returns A dict with the driver_info values.

Raises MissingParameterValue, if any of the required parameters are missing.

ironic.drivers.modules.ilo.boot.**prepare_node_for_deploy**(*task*)

Common preparatory steps for all iLO drivers.

This method performs common preparatory steps required for all drivers. 1. Power off node 2. Disables secure boot, if it is in enabled state. 3. Updates boot_mode capability to uefi if secure boot is requested. 4. Changes boot mode of the node if secure boot is disabled currently.

Parameters **task** a TaskManager instance containing the node to act on.

Raises IloOperationError, if some operation on iLO failed.

ironic.drivers.modules.ilo.common module

Common functionalities shared between different iLO modules.

ironic.drivers.modules.ilo.common.**POST_FINISHEDPOST_STATE = 'FinishedPost'**

Node is in FinishedPost post state.

ironic.drivers.modules.ilo.common.**POST_INPOSTDISCOVERY_STATE = 'InPostDiscoveryComplete'**

Node is in InPostDiscoveryComplete post state.

ironic.drivers.modules.ilo.common.**POST_INPOST_STATE = 'InPost'**

Node is in InPost post state.

ironic.drivers.modules.ilo.common.**POST_NULL_STATE = 'Null'**

Node is in Null post state.

ironic.drivers.modules.ilo.common.**POST_POWEROFF_STATE = 'PowerOff'**

Node is in PowerOff post state.

ironic.drivers.modules.ilo.common.**POST_RESET_STATE = 'Reset'**
Node is in Reset post state.

ironic.drivers.modules.ilo.common.**POST_UNKNOWN_STATE = 'Unknown'**
Node is in Unknown post state.

ironic.drivers.modules.ilo.common.**SUPPORTED_BOOT_MODE_LEGACY_BIOS_AND_UEFI = 'legacy bios and uefi'**
Node supports both legacy BIOS and UEFI boot mode.

ironic.drivers.modules.ilo.common.**SUPPORTED_BOOT_MODE_LEGACY_BIOS_ONLY = 'legacy bios only'**
Node supports only legacy BIOS boot mode.

ironic.drivers.modules.ilo.common.**SUPPORTED_BOOT_MODE_UEFI_ONLY = 'uefi only'**
Node supports only UEFI boot mode.

ironic.drivers.modules.ilo.common.**add_certificates**(*task*, *cert_file_list=None*)
Adds certificates to the node.

Adds certificates to the node based on the driver info provided.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **cert_file_list** List of certificates to be added to the node. If None, certificates from path configured in webserver_verify_ca will be added to the node.

Raises IloOperationError on an error from IloClient library.

Raises IloOperationNotSupported if retrieving post state is not supported on the server.

Raises InvalidParameterValue, if any of the required parameters are invalid.

ironic.drivers.modules.ilo.common.**attach_vmedia**(*node*, *device*, *url*)
Attaches the given url as virtual media on the node.

Parameters

- **node** an ironic node object.
- **device** the virtual media device to attach
- **url** the http/https url to attach as the virtual media device

Raises IloOperationError if insert virtual media failed.

ironic.drivers.modules.ilo.common.**cleanup_vmedia_boot**(*task*)
Cleans a node after a virtual media boot.

This method cleans up a node after a virtual media boot. It deletes the floppy image if it exists in CONF.ilo.swift_ilo_container or web server. It also ejects both virtual media cdrom and virtual media floppy.

Parameters **task** a TaskManager instance containing the node to act on.

ironic.drivers.modules.ilo.common.**clear_certificates**(*task*, *cert_file_list=None*)
Clears any certificates added to the node.

Clears the certificates added to the node as part of any Ironic operation

Parameters

- **task** a TaskManager instance containing the node to act on.
- **cert_file_list** List of certificates to be removed from node. If None, all the certificates present on the node will be removed.

Raises IloOperationError on an error from IloClient library.

Raises IloOperationNotSupported if retrieving post state is not supported on the server.

ironic.drivers.modules.ilo.common.**copy_image_to_swift**(*source_file_path*, *destination_object_name*)

Uploads the given image to swift.

This method copies the given image to swift.

Parameters

- **source_file_path** The absolute path of the image file which needs to be copied to swift.
- **destination_object_name** The name of the object that will contain the copied image.

Raises SwiftOperationError, if any operation with Swift fails.

Returns temp url from swift after the source image is uploaded.

ironic.drivers.modules.ilo.common.**copy_image_to_web_server**(*source_file_path*, *destination*)

Copies the given image to the http web server.

This method copies the given image to the http_root location. It enables read-write access to the image else the deploy fails as the image file at the web_server url is inaccessible.

Parameters

- **source_file_path** The absolute path of the image file which needs to be copied to the web server root.
- **destination** The name of the file that will contain the copied image.

Raises ImageUploadFailed exception if copying the source file to the web server fails.

Returns image url after the source image is uploaded.

ironic.drivers.modules.ilo.common.**destroy_floppy_image_from_web_server**(*node*)

Removes the temporary floppy image.

It removes the floppy image created for deploy. :param node: an ironic node object.

ironic.drivers.modules.ilo.common.**eject_vmedia_devices**(*task*)

Ejects virtual media devices.

This method ejects virtual media floppy and cdrom.

Parameters **task** a TaskManager instance containing the node to act on.

Returns None

Raises IloOperationError, if some error was encountered while trying to eject virtual media floppy or cdrom.

ironic.drivers.modules.ilo.common.**get_current_boot_mode**(*node*)

Get the current boot mode for a node.

Parameters **node** an ironic node object.

Raises IloOperationError if failed to fetch boot mode.

Raises IloOperationNotSupported if node does not support getting pending boot mode.

ironic.drivers.modules.ilo.common.**get_ilo_object**(*node*)
Gets an IloClient object from proliantutils library.

Given an ironic node object, this method gives back a IloClient object to do operations on the iLO.

Parameters **node** an ironic node object.

Returns an IloClient object.

Raises InvalidParameterValue on invalid inputs.

Raises MissingParameterValue if some mandatory information is missing on the node

ironic.drivers.modules.ilo.common.**get_secure_boot_mode**(*task*)
Retrieves current enabled state of UEFI secure boot on the node

Returns the current enabled state of UEFI secure boot on the node.

Parameters **task** a task from TaskManager.

Raises MissingParameterValue if a required iLO parameter is missing.

Raises IloOperationError on an error from IloClient library.

Raises IloOperationNotSupported if UEFI secure boot is not supported.

Returns Boolean value indicating current state of UEFI secure boot on the node.

ironic.drivers.modules.ilo.common.**get_server_post_state**(*node*)
Get the current state of system POST.

Parameters **node** an ironic node object.

Returns POST state of the server. The valida states are:- null, Unknown, Reset, PowerOff, InPost, InPostDiscoveryComplete and FinishedPost.

Raises IloOperationError on an error from IloClient library.

Raises IloOperationNotSupported if retrieving post state is not supported on the server.

ironic.drivers.modules.ilo.common.**parse_driver_info**(*node*)
Gets the driver specific Node info.

This method validates whether the driver_info property of the supplied node contains the required information for this driver.

Parameters **node** an ironic Node object.

Returns a dict containing information from driver_info (or where applicable, config values).

Raises InvalidParameterValue if any parameters are incorrect

Raises MissingParameterValue if some mandatory information is missing on the node

ironic.drivers.modules.ilo.common.**remove_image_from_swift**(*object_name*, *associated_with=None*)
Removes the given image from swift.

This method removes the given image name from swift. It deletes the image if it exists in CONF.ilo.swift_ilo_container

Parameters

- **object_name** The name of the object which needs to be removed from swift.
- **associated_with** string to depict the component/operation this object is associated to.

ironic.drivers.modules.ilo.common.**remove_image_from_web_server**(*object_name*)
Removes the given image from the configured web server.

This method removes the given image from the http_root location, if the image exists.

Parameters **object_name** The name of the image file which needs to be removed from the web server root.

ironic.drivers.modules.ilo.common.**remove_single_or_list_of_files**(*file_location*)
Removes (deletes) the file or list of files.

This method only accepts single or list of files to delete. If single file is passed, this method removes (deletes) the file. If list of files is passed, this method removes (deletes) each of the files iteratively.

Parameters **file_location** a single or a list of file paths

ironic.drivers.modules.ilo.common.**set_boot_mode**(*node*, *boot_mode*)
Sets the node to boot using boot_mode for the next boot.

Parameters

- **node** an ironic node object.
- **boot_mode** Next boot mode.

Raises IloOperationError if setting boot mode failed.

ironic.drivers.modules.ilo.common.**set_secure_boot_mode**(*task*, *flag*)
Enable or disable UEFI Secure Boot for the next boot

Enable or disable UEFI Secure Boot for the next boot

Parameters

- **task** a task from TaskManager.
- **flag** Boolean value. True if the secure boot to be enabled in next boot.

Raises IloOperationError on an error from IloClient library.

Raises IloOperationNotSupported if UEFI secure boot is not supported.

ironic.drivers.modules.ilo.common.**setup_uefi_https**(*task*, *iso*, *persistent=False*)
Sets up system to boot from UEFIHTTP boot device.

Sets the one-time/persistent boot device to UEFIHTTP based on the argument supplied.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **iso** ISO URL to be set to boot from.
- **persistent** Indicates whether the system should be set to boot from the given device one-time or each time.

Raises IloOperationError on an error from IloClient library.

Raises IloOperationNotSupported if retrieving post state is not supported on the server.

ironic.drivers.modules.ilo.common.**setup_vmedia**(*task*, *iso*, *ramdisk_options=None*)

Attaches virtual media and sets it as boot device.

This method attaches the given bootable ISO as virtual media, prepares the arguments for ramdisk in virtual media floppy.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **iso** a bootable ISO image href to attach to. Should be either of below:
  - A Swift object - It should be of format `swift:<object-name>`. It is assumed that the image object is present in CONF.ilo.swift_ilo_container;
  - A Glance image - It should be format `glance://<glance-image-uuid>` or just `<glance-image-uuid>`;
  - An HTTP URL.
- **ramdisk_options** the options to be passed to the ramdisk in virtual media floppy.

Raises ImageCreationFailed, if it failed while creating the floppy image.

Raises IloOperationError, if some operation on iLO failed.

ironic.drivers.modules.ilo.common.**setup_vmedia_for_boot**(*task*, *boot_iso*, *parameters=None*)

Sets up the node to boot from the given ISO image.

This method attaches the given boot_iso on the node and passes the required parameters to it via virtual floppy image.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **boot_iso** a bootable ISO image to attach to. Should be either of below:
  - A Swift object - It should be of format `swift:<object-name>`. It is assumed that the image object is present in CONF.ilo.swift_ilo_container;
  - A Glance image - It should be format `glance://<glance-image-uuid>` or just `<glance-image-uuid>`;
  - An HTTP URL.
- **parameters** the parameters to pass in the virtual floppy image in a dictionary. This is optional.

Raises ImageCreationFailed, if it failed while creating the floppy image.

Raises SwiftOperationError, if any operation with Swift fails.

Raises IloOperationError, if attaching virtual media failed.

ironic.drivers.modules.ilo.common.**update_boot_mode**(*task*)

Update instance_info with boot mode to be used for deploy.

This method updates instance_info with boot mode to be used for deploy if node properties[capabilities] do not have boot_mode. It sets the boot mode on the node.

Parameters **task** Task object.

Raises IloOperationError if setting boot mode failed.

ironic.drivers.modules.ilo.common.**update_ipmi_properties**(*task*)
Update ipmi properties to node driver_info

Parameters **task** a task from TaskManager.

ironic.drivers.modules.ilo.common.**validate_security_parameter_values**(*sec_param_info*)
Validate security parameter with valid values.

Parameters **sec_param_info** dict object containing the security parameter info

Raises MissingParameterValue, for missing fields (or values) in security parameter info.

Raises InvalidParameterValue, for unsupported security parameter

Returns tuple of security param, ignore and enable parameters.

ironic.drivers.modules.ilo.common.**verify_image_checksum**(*image_location*, *expected_checksum*)
Verifies checksum (md5) of image file against the expected one.

This method generates the checksum of the image file on the fly and verifies it against the expected checksum provided as argument.

Parameters

- **image_location** location of image file whose checksum is verified.
- **expected_checksum** checksum to be checked against

Raises ImageRefValidationFailed, if invalid file path or verification fails.

ironic.drivers.modules.ilo.console module

iLO Deploy Driver(s) and supporting methods.

class ironic.drivers.modules.ilo.console.**IloConsoleInterface**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.ipmitool.IPMIShellinaboxConsole*

A ConsoleInterface that uses ipmitool and shellinabox.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

validate(*task*)
Validate the Node console info.

Parameters **task** a task from TaskManager.

Raises InvalidParameterValue

Raises MissingParameterValue when a required parameter is missing

ironic.drivers.modules.ilo.firmware_processor module

Firmware file processor

class ironic.drivers.modules.ilo.firmware_processor.**FirmwareImageLocation**(*fw_image_location*, *fw_image_filename*)

Bases: object

Firmware image location class

This class acts as a wrapper class for the firmware image location. It primarily helps in removing the firmware files from their respective locations, made available for firmware update operation.

remove()

Exposed method to remove the wrapped firmware file

This method gets overridden by the remove method for the respective type of firmware file location it wraps.

class ironic.drivers.modules.ilo.firmware_processor.**FirmwareProcessor**(*url*)

Bases: object

Firmware file processor

This class helps in downloading the firmware file from url, extracting the firmware file (if its in compact format) and makes it ready for firmware update operation. In future, methods can be added as and when required to extend functionality for different firmware file types.

process_fw_on(*node*, *expected_checksum*)

Processes the firmware file from the url

This is the template method which downloads the firmware file from url, verifies checksum and extracts the firmware and makes it ready for firmware update operation. `_download_fw_to` method is set in the firmware processor object creation factory method, `get_fw_processor()`, based on the url type. :param node: a single Node. :param expected_checksum: checksum to be checked against. :returns: wrapper object of raw firmware image location :raises: IloOperationError, on failure to process firmware file. :raises: ImageDownloadFailed, on failure to download the original file. :raises: ImageRefValidationFailed, on failure to verify the checksum. :raises: SwiftOperationError, if upload to Swift fails. :raises: ImageUploadFailed, if upload to web server fails.

ironic.drivers.modules.ilo.firmware_processor.**get_and_validate_firmware_image_info**(*firmware_i*, *firmware_u*)

Validates the firmware image info and returns the retrieved values.

Parameters **firmware_image_info** dict object containing the firmware image info

Raises MissingParameterValue, for missing fields (or values) in image info.

Raises InvalidParameterValue, for unsupported firmware component

Returns tuple of firmware url, checksum, component when the firmware update is ilo based.

ironic.drivers.modules.ilo.firmware_processor.**get_swift_url**(*parsed_url*)

Gets swift temp url.

It generates a temp url for the swift based firmware url to the target file. Expecting url as swift://containername/objectname.

Parameters **parsed_url** Parsed url object.

Raises SwiftOperationError, on failure to get url from swift.

ironic.drivers.modules.ilo.firmware_processor.**verify_firmware_update_args**(*func*)
Verifies the firmware update arguments.

ironic.drivers.modules.ilo.inspect module

iLO Inspect Interface

class ironic.drivers.modules.ilo.inspect.**IloInspect**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.InspectInterface*

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

inspect_hardware(*task*)
Inspect hardware to get the hardware properties.

Inspects hardware to get the essential and additional hardware properties. It fails if any of the essential properties are not received from the node. It doesnt fail if node fails to return any capabilities as the capabilities differ from hardware to hardware mostly.

Parameters **task** a TaskManager instance.

Raises HardwareInspectionFailure if essential properties could not be retrieved successfully.

Raises IloOperationError if system fails to get power state.

Returns The resulting state of inspection.

validate(*task*)
Check that driver_info contains required ILO credentials.

Validates whether the driver_info property of the supplied tasks node contains the required credentials information.

Parameters **task** a task from TaskManager.

Raises InvalidParameterValue if required iLO parameters are not valid.

Raises MissingParameterValue if a required parameter is missing.

ironic.drivers.modules.ilo.management module

iLO Management Interface

class ironic.drivers.modules.ilo.management.**Ilo5Management**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.ilo.management.IloManagement*

clear_ca_certificates(*task*, *certificate_files*)
Clears the certificates provided in the list of files to iLO.

Parameters

- **task** a task from TaskManager.
- **certificate_files** a list of cerificate files.

Raises NodeCleaningFailure, on failure to execute of clean step.

Raises InstanceDeployFailure, on failure to execute of deploy step.

erase_devices(*task*, ***kwargs*)
Erase all the drives on the node.

This method performs out-of-band sanitize disk erase on all the supported physical drives in the node. This erase cannot be performed on logical drives.

Parameters **task** a TaskManager instance.

Raises InvalidParameterValue, if any of the arguments are invalid.

Raises IloError on an error from iLO.

one_button_secure_erase(*task*)
Erase the whole system securely.

The One-button secure erase process resets iLO and deletes all licenses stored there, resets BIOS settings, and deletes all Active Health System (AHS) and warranty data stored on the system. It also erases supported non-volatile storage data and deletes any deployment setting profiles.

Parameters **task** a TaskManager instance.

Raises IloError on an error from iLO.

class ironic.drivers.modules.ilo.management.**IloManagement**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.ManagementInterface*

activate_license(*task*, ***kwargs*)
Activates iLO Advanced license.

Parameters **task** a TaskManager object.

Raises InvalidParameterValue, if any of the arguments are invalid.

Raises NodeCleaningFailure, on failure to execute of clean step.

clear_iscsi_boot_target(*task*)
Unset iSCSI details of the system in UEFI boot mode.

Parameters **task** a task from TaskManager.

Raises IloCommandNotSupportedInBiosError if system in BIOS boot mode.

Raises IloError on an error from iLO.

clear_secure_boot_keys(*task*)
Clear all secure boot keys.

Clears all the secure boot keys. This operation is supported only on HP Proliant Gen9 and above servers.

Parameters **task** a task from TaskManager.

Raises NodeCleaningFailure, on failure to execute of clean step.

Raises InstanceDeployFailure, on failure to execute of deploy step.

flash_firmware_sum(*task*, ***kwargs*)
Deploy step to Update the firmware using Smart Update Manager (SUM).

Parameters **task** a TaskManager object.

Raises InstanceDeployFailure, on failure to execute of deploy step.

Returns states.DEPLOYWAIT to signify the step will be completed async

get_boot_device(*task*)
Get the current boot device for a node.

Returns the current boot device of the node.

Parameters **task** a task from TaskManager.

Raises MissingParameterValue if a required iLO parameter is missing.

Raises IloOperationError on an error from IloClient library.

Returns

a dictionary containing:

boot_device the boot device, one of the supported devices listed in *ironic.common.boot_devices* or None if it is unknown.

persistent Whether the boot device will persist to all future boots or not, None if it is unknown.

get_boot_mode(*task*)
Get the current boot mode for a node.

Provides the current boot mode of the node.

Parameters **task** A task from TaskManager.

Raises IloOperationError on an error from IloClient library.

Returns The boot mode, one of `ironic.common.boot_mode` or None if it is unknown.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

get_secure_boot_state(*task*)
Get the current secure boot state for the node.

Parameters **task** A task from TaskManager.

Raises MissingParameterValue if a required parameter is missing

Raises IloOperationError on an error from IloClient library.

Raises UnsupportedDriverExtension if secure boot is not supported by the hardware

Returns Boolean

get_sensors_data(*task*)
Get sensors data.

Parameters **task** a TaskManager instance.

Raises FailedToGetSensorData when getting the sensor data fails.

Raises FailedToParseSensorData when parsing sensor data fails.

Raises InvalidParameterValue if required ipmi parameters are missing.

Raises MissingParameterValue if a required parameter is missing.

Returns returns a dict of sensor data group by sensor type.

`get_supported_boot_devices`(*task*)
Get a list of the supported boot devices.

Parameters **task** a task from TaskManager.

Returns A list with the supported boot devices defined in *ironic.common.boot_devices*.

`get_supported_boot_modes`(*task*)
Get a list of the supported boot devices.

Parameters **task** a task from TaskManager.

Raises IloOperationError if any exception happens in proliantutils

Returns A list with the supported boot devices defined in *ironic.common.boot_devices*.

`inject_nmi`(*task*)
Inject NMI, Non Maskable Interrupt.

Inject NMI (Non Maskable Interrupt) for a node immediately.

Parameters **task** A TaskManager instance containing the node to act on.

Raises IloCommandNotSupportedError if system does not support NMI injection.

Raises IloError on an error from iLO.

Returns None

`reset_bios_to_default`(*task*)
Resets the BIOS settings to default values.

Resets BIOS to default settings. This operation is currently supported only on HP Proliant Gen9 and above servers.

Parameters **task** a task from TaskManager.

Raises NodeCleaningFailure, on failure to execute of clean step.

Raises InstanceDeployFailure, on failure to execute of deploy step.

`reset_ilo`(*task*)
Resets the iLO.

Parameters **task** a task from TaskManager.

Raises NodeCleaningFailure, on failure to execute of clean step.

Raises InstanceDeployFailure, on failure to execute of deploy step.

`reset_ilo_credential`(*task*, *change_password=None*)
Resets the iLO password.

Parameters

- **task** a task from TaskManager.
- **change_password** Value for password to update on iLO.

Raises NodeCleaningFailure, on failure to execute of clean step.

Raises InstanceDeployFailure, on failure to execute of deploy step.

reset_secure_boot_keys_to_default(*task*)
Reset secure boot keys to manufacturing defaults.

Resets the secure boot keys to manufacturing defaults. This operation is supported only on HP Proliant Gen9 and above servers.

Parameters **task** a task from TaskManager.

Raises NodeCleaningFailure, on failure to execute of clean step.

Raises InstanceDeployFailure, on failure to execute of deploy step.

security_parameters_update(*task*, ***kwargs*)
Updates the security parameters.

Parameters **task** a TaskManager object.

set_boot_device(*task*, *device*, *persistent=False*)
Set the boot device for a node.

Set the boot device to use on next reboot of the node.

Parameters

- **task** a task from TaskManager.
- **device** the boot device, one of the supported devices listed in *ironic.common.boot_devices*.
- **persistent** Boolean value. True if the boot device will persist to all future boots, False if not. Default: False.

Raises InvalidParameterValue if an invalid boot device is specified.

Raises MissingParameterValue if a required parameter is missing.

Raises IloOperationError on an error from IloClient library.

set_boot_mode(*task*, *mode*)
Set the boot mode for a node.

Set the boot mode to use on next reboot of the node.

Parameters

- **task** A task from TaskManager.
- **mode** The boot mode, one of *ironic.common.boot_modes*.

Raises InvalidParameterValue if an invalid boot mode is specified.

Raises IloOperationError if setting boot mode failed.

set_iscsi_boot_target(*task*)
Set iSCSI details of the system in UEFI boot mode.

The initiator is set with the target details like IQN, LUN, IP, Port etc. :param task: a task from TaskManager. :raises: MissingParameterValue if a required parameter is missing. :raises: IloCommandNotSupportedInBiosError if system in BIOS boot mode. :raises: IloError on an error from iLO.

set_secure_boot_state(*task*, *state*)
Set the current secure boot state for the node.

Parameters

- **task** A task from TaskManager.
- **state** A new state as a boolean.

Raises MissingParameterValue if a required parameter is missing

Raises IloOperationError on an error from IloClient library.

Raises UnsupportedDriverExtension if secure boot is not supported by the hardware

update_auth_failure_logging_threshold(*task*, ***kwargs*)
Updates the Auth Failure Logging Threshold security parameter.

Parameters **task** a TaskManager object.

update_firmware(*task*, ***kwargs*)
Updates the firmware.

Parameters **task** a TaskManager object.

Raises InvalidParameterValue if update firmware mode is not ilo. Even applicable for invalid input cases.

Raises NodeCleaningFailure, on failure to execute of clean step.

Raises InstanceDeployFailure, on failure to execute of deploy step.

update_firmware_sum(*task*, ***kwargs*)
Clean step to update the firmware using Smart Update Manager (SUM)

Parameters **task** a TaskManager object.

Raises NodeCleaningFailure, on failure to execute of clean step.

Returns states.CLEANWAIT to signify the step will be completed async

update_minimum_password_length(*task*, ***kwargs*)
Updates the Minimum Password Length security parameter.

Parameters **task** a TaskManager object.

validate(*task*)
Check that driver_info contains required ILO credentials.

Validates whether the driver_info property of the supplied tasks node contains the required credentials information.

Parameters **task** a task from TaskManager.

Raises InvalidParameterValue if required iLO parameters are not valid.

Raises MissingParameterValue if a required parameter is missing.

ironic.drivers.modules.ilo.power module

iLO Power Driver

class ironic.drivers.modules.ilo.power.**IloPower**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.PowerInterface*

get_power_state(*task*)
Gets the current power state.

Parameters

- **task** a TaskManager instance.
- **node** The Node.

Returns one of *ironic.common.states* POWER_OFF, POWER_ON or ERROR.

Raises InvalidParameterValue if required iLO credentials are missing.

Raises IloOperationError on an error from IloClient library.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

get_supported_power_states(*task*)
Get a list of the supported power states.

Parameters **task** A TaskManager instance containing the node to act on. currently not used.

Returns A list with the supported power states defined in *ironic.common.states*.

reboot(*task*, *timeout=None*)
Reboot the node

Parameters

- **task** a TaskManager instance.
- **timeout** timeout (in seconds). Unsupported by this interface.

Raises PowerStateFailure if the final state of the node is not POWER_ON.

Raises IloOperationError on an error from IloClient library.

set_power_state(*task*, *power_state*, *timeout=None*)
Turn the current power state on or off.

Parameters

- **task** a TaskManager instance.
- **power_state** The desired power state POWER_ON,POWER_OFF or REBOOT from *ironic.common.states*.

- **timeout** timeout (in seconds). Unsupported by this interface.

Raises InvalidParameterValue if an invalid power state was specified.

Raises IloOperationError on an error from IloClient library.

Raises PowerStateFailure if the power couldnt be set to power_state.

validate(*task*)

Check if node.driver_info contains the required iLO credentials.

Parameters

- **task** a TaskManager instance.
- **node** Single node object.

Raises InvalidParameterValue if required iLO credentials are missing.

ironic.drivers.modules.ilo.raid module

iLO5 RAID specific methods

class ironic.drivers.modules.ilo.raid.**Ilo5RAID**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.RAIDInterface*

Implementation of OOB RAIDInterface for iLO5.

apply_configuration(*task*, *raid_config*, *create_root_volume=True*, *create_nonroot_volumes=False*)

Applies RAID configuration on the given node.

Parameters

- **task** A TaskManager instance.
- **raid_config** The RAID configuration to apply.
- **create_root_volume** Setting this to False indicates not to create root volume that is specified in raid_config. Default value is True.
- **create_nonroot_volumes** Setting this to False indicates not to create nonroot volumes (all except the root volume) in raid_config. Default value is True.
- **delete_existing** Setting this to True indicates to delete RAID configuration prior to creating the new configuration.

Raises InvalidParameterValue, if the RAID configuration is invalid.

Returns states.DEPLOYWAIT if RAID configuration is in progress asynchronously or None if it is complete.

create_configuration(*task*, *create_root_volume=True*, *create_nonroot_volumes=True*)

Create a RAID configuration on a bare metal using agent ramdisk.

This method creates a RAID configuration on the given node.

Parameters

- **task** a TaskManager instance.

- **create_root_volume** If True, a root volume is created during RAID configuration. Otherwise, no root volume is created. Default is True.
- **create_nonroot_volumes** If True, non-root volumes are created. If False, no non-root volumes are created. Default is True.

Raises MissingParameterValue, if node.target_raid_config is missing or was found to be empty after skipping root volume and/or non-root volumes.

Raises NodeCleaningFailure, on failure to execute clean step.

Raises InstanceDeployFailure, on failure to execute deploy step.

delete_configuration(*task*)
Delete the RAID configuration.

Parameters **task** a TaskManager instance containing the node to act on.

Raises NodeCleaningFailure, on failure to execute clean step.

Raises InstanceDeployFailure, on failure to execute deploy step.

get_properties()
Return the properties of the interface.

ironic.drivers.modules.ilo.vendor module

Vendor Interface for iLO drivers and its supporting methods.

class ironic.drivers.modules.ilo.vendor.**VendorPassthru**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.VendorInterface*

Vendor-specific interfaces for iLO deploy drivers.

boot_into_iso(*task*, ***kwargs*)
Attaches an ISO image in glance and reboots bare metal.

This method accepts an ISO image href (a Glance UUID or an HTTP(S) URL) attaches it as virtual media and then reboots the node. This is useful for debugging purposes. This can be invoked only when the node is in manage state.

Parameters

- **task** A TaskManager object.
- **kwargs** The arguments sent with vendor passthru. The expected kwargs are:

```
'boot_iso_href': href of the image to be booted. This␣
↪can be
    a Glance UUID or an HTTP(S) URL.
```

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

validate(*task*, *method*, ***kwargs*)
Validate vendor-specific actions.

Checks if a valid vendor passthru method was passed and validates the parameters for the vendor passthru method.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **method** method to be validated.
- **kwargs** kwargs containing the vendor passthru methods parameters.

Raises MissingParameterValue, if some required parameters were not passed.

Raises InvalidParameterValue, if any of the parameters have invalid value.

Module contents

ironic.drivers.modules.intel_ipmi package

Submodules

ironic.drivers.modules.intel_ipmi.management module

Intel IPMI Hardware.

Supports Intel Speed Select Performance Profile.

class ironic.drivers.modules.intel_ipmi.management.**IntelIPMIManagement**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.ipmitool.IPMIManagement*

configure_intel_speedselect(*task*, ***kwargs*)

Module contents

ironic.drivers.modules.irmc package

Submodules

ironic.drivers.modules.irmc.bios module

iRMC BIOS configuration specific methods

class ironic.drivers.modules.irmc.bios.**IRMCBIOS**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.BIOSInterface*

apply_configuration(*task*, *settings*)

Applies BIOS configuration on the given node.

This method takes the BIOS settings from the settings param and applies BIOS configuration on the given node. After the BIOS configuration is done, self.cache_bios_settings() may be called to sync the nodes BIOS-related information with the BIOS configuration applied on the node. It will also validate the given settings before applying any settings and manage

failures when setting an invalid BIOS config. In the case of needing password to update the BIOS config, it will be taken from the driver_info properties.

Parameters

- **task** a TaskManager instance.
- **settings** Dictionary containing the BIOS configuration. It may be an empty dictionary as well.

Raises IRMCOperationError,if apply bios settings failed.

cache_bios_settings(*task*)
Store or update BIOS settings on the given node.

This method stores BIOS properties to the bios settings db

Parameters **task** a TaskManager instance.

Raises IRMCOperationError,if get bios settings failed.

Returns None if it is complete.

factory_reset(*task*)
Reset BIOS configuration to factory default on the given node.

Parameters **task** a TaskManager instance.

Raises UnsupportedDriverExtension, if the nodes driver doesnt support BIOS reset.

get_properties()
Return the properties of the interface.

validate(*task*)
Validate the driver-specific Node info.

This method validates whether the driver_info property of the supplied node contains the required information for this driver to manage the BIOS settings of the node.

Parameters **task** a TaskManager instance containing the node to act on.

Raises InvalidParameterValue if required driver_info attribute is missing or invalid on the node.

Raises MissingParameterValue if a required parameter is missing in the driver_info property.

ironic.drivers.modules.irmc.boot module

iRMC Boot Driver

class ironic.drivers.modules.irmc.boot.**IRMCPXEBoot**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.pxe.PXEBoot*

iRMC PXE boot.

prepare_ramdisk(*task*, *ramdisk_params*)
Prepares the boot of Ironic ramdisk using PXE.

This method prepares the boot of the deploy kernel/ramdisk after reading relevant information from the nodes driver_info and instance_info.

Parameters

- **task** a task from TaskManager.
- **ramdisk_params** the parameters to be passed to the ramdisk. pxe driver passes these parameters as kernel command-line arguments.

Returns None

Raises MissingParameterValue, if some information is missing in nodes driver_info or instance_info.

Raises InvalidParameterValue, if some information provided is invalid.

Raises IronicException, if some power or set boot device operation failed on the node.

class ironic.drivers.modules.irmc.boot.**IRMCVirtualMediaBoot**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.BootInterface*, *ironic.drivers.modules.irmc.boot.IRMCVolumeBootMixIn*

iRMC Virtual Media boot-related actions.

capabilities = ['iscsi_volume_boot', 'fibre_channel_volume_boot']

clean_up_instance(*task*)

Cleans up the boot of instance.

This method cleans up the environment that was setup for booting the instance.

Parameters **task** a task from TaskManager.

Returns None

Raises IRMCOperationError if iRMC operation failed.

clean_up_ramdisk(*task*)

Cleans up the boot of ironic ramdisk.

This method cleans up the environment that was setup for booting the deploy or rescue ramdisk.

Parameters **task** a task from TaskManager.

Returns None

Raises IRMCOperationError if iRMC operation failed.

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

prepare_instance(*task*)

Prepares the boot of instance.

This method prepares the boot of the instance after reading relevant information from the nodes database.

Parameters **task** a task from TaskManager.

Returns None

prepare_ramdisk(*task*, *ramdisk_params*)
Prepares the deploy or rescue ramdisk using virtual media.

Prepares the options for the deploy or rescue ramdisk, sets the node to boot from virtual media cdrom.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **ramdisk_params** the options to be passed to the ramdisk.

Raises ImageRefValidationFailed if no image service can handle specified href.

Raises ImageCreationFailed, if it failed while creating the floppy image.

Raises InvalidParameterValue if the validation of the PowerInterface or ManagementInterface fails.

Raises IRMCOperationError, if some operation on iRMC fails.

validate(*task*)
Validate the deployment information for the tasks node.

Parameters **task** a TaskManager instance containing the node to act on.

Raises InvalidParameterValue, if config option has invalid value.

Raises IRMCSharedFileSystemNotMounted, if shared file system is not mounted.

Raises InvalidParameterValue, if some information is invalid.

Raises MissingParameterValue if kernel_id and ramdisk_id are missing in the Glance image, or if kernel and ramdisk are missing in the Non Glance image.

validate_rescue(*task*)
Validate that the node has required properties for rescue.

Parameters **task** a TaskManager instance with the node being checked

Raises MissingParameterValue if node is missing one or more required parameters

Raises InvalidParameterValue, if any of the parameters have invalid value.

class ironic.drivers.modules.irmc.boot.**IRMCVolumeBootMixIn**
Bases: object

Mix-in class for volume boot configuration to iRMC

iRMC has a feature to set up remote boot to a server. This feature can be used by VIOM (Virtual I/O Manager) library of SCCI client.

ironic.drivers.modules.irmc.boot.**attach_boot_iso_if_needed**(*task*)
Attaches boot ISO for a deployed node if it exists.

This method checks the instance info of the bare metal node for a boot ISO. If the instance info has a value of key boot_iso, it indicates that boot_option is netboot. Threfore it attaches the boot ISO on the bare metal node and then sets the node to boot from virtual media cdrom.

Parameters **task** a TaskManager instance containing the node to act on.

Raises IRMCOperationError if attaching virtual media failed.

Raises InvalidParameterValue if the validation of the ManagementInterface fails.

ironic.drivers.modules.irmc.boot.**check_share_fs_mounted**()
Check if Share File System (NFS or CIFS) is mounted.

Raises InvalidParameterValue, if config option has invalid value.

Raises IRMCSharedFileSystemNotMounted, if shared file system is not mounted.

ironic.drivers.modules.irmc.common module

Common functionalities shared between different iRMC modules.

ironic.drivers.modules.irmc.common.**get_irmc_client**(*node*)
Gets an iRMC SCCI client.

Given an ironic node object, this method gives back a iRMC SCCI client to do operations on the iRMC.

Parameters **node** An ironic node object.

Returns scci_cmd partial function which takes a SCCI command param.

Raises InvalidParameterValue on invalid inputs.

Raises MissingParameterValue if some mandatory information is missing on the node

Raises IRMCOperationError if iRMC operation failed

ironic.drivers.modules.irmc.common.**get_irmc_report**(*node*)
Gets iRMC SCCI report.

Given an ironic node object, this method gives back a iRMC SCCI report.

Parameters **node** An ironic node object.

Returns A xml.etree.ElementTree object.

Raises InvalidParameterValue on invalid inputs.

Raises MissingParameterValue if some mandatory information is missing on the node.

Raises scci.SCCIInvalidInputError if required parameters are invalid.

Raises scci.SCCIClientError if SCCI failed.

ironic.drivers.modules.irmc.common.**get_secure_boot_mode**(*node*)
Get the current secure boot mode.

Parameters **node** An ironic node object.

Raises UnsupportedDriverExtension if secure boot is not present.

Raises IRMCOperationError if the operation fails.

ironic.drivers.modules.irmc.common.**parse_driver_info**(*node*)
Gets the specific Node driver info.

This method validates whether the driver_info property of the supplied node contains the required information for this driver.

Parameters **node** An ironic node object.

Returns A dict containing information from driver_info and default values.

Raises InvalidParameterValue if invalid value is contained in the driver_info property.

Raises MissingParameterValue if some mandatory key is missing in the driver_info property.

ironic.drivers.modules.irmc.common.**set_secure_boot_mode**(*node*, *enable*)
Enable or disable UEFI Secure Boot

Parameters

- **node** An ironic node object.
- **enable** Boolean value. True if the secure boot to be enabled.

Raises IRMCOperationError if the operation fails.

ironic.drivers.modules.irmc.common.**update_ipmi_properties**(*task*)
Update ipmi properties to node driver_info.

Parameters **task** A task from TaskManager.

ironic.drivers.modules.irmc.inspect module

iRMC Inspect Interface

class ironic.drivers.modules.irmc.inspect.**IRMCInspect**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.InspectInterface*

Interface for out of band inspection.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

inspect_hardware(*task*)
Inspect hardware.

Inspect hardware to obtain the essential hardware properties and mac addresses.

Parameters **task** a task from TaskManager.

Raises HardwareInspectionFailure, if hardware inspection failed.

Returns states.MANAGEABLE, if hardware inspection succeeded.

validate(*task*)
Validate the driver-specific inspection information.

This method validates whether the driver_info property of the supplied node contains the required information for this driver.

Parameters **task** a TaskManager instance containing the node to act on.

Raises InvalidParameterValue if required driver_info attribute is missing or invalid on the node.

Raises MissingParameterValue if a required parameter is missing.

ironic.drivers.modules.irmc.inspect.**METRICS** =
<**ironic_lib.metrics.NoopMetricLogger object**>

SC2.mib: sc2UnitNodeClass returns NIC type.

sc2UnitNodeClass OBJECT-TYPE SYNTAX INTEGER { unknown(1), primary(2), secondary(3), management-blade(4), secondary-remote(5), secondary-remote-backup(6), baseboard-controller(7) } ACCESS read-only STATUS mandatory DESCRIPTION Management node class: primary: local operating system interface secondary: local management controller LAN interface management-blade: management blade interface (in a blade server chassis) secondary-remote: remote management controller (in an RSB concentrator environment) secondary-remote-backup: backup remote management controller baseboard-controller: local baseboard management controller (BMC) ::= { sc2ManagementNodes 8 }

ironic.drivers.modules.irmc.inspect.**NODE_CLASS_OID** =
'1.3.6.1.4.1.231.2.10.2.2.10.3.1.1.8.1'

SC2.mib: sc2UnitNodeMacAddress returns NIC MAC address

sc2UnitNodeMacAddress OBJECT-TYPE SYNTAX PhysAddress ACCESS read-only STATUS mandatory DESCRIPTION Management node hardware (MAC) address ::= { sc2ManagementNodes 9 }

ironic.drivers.modules.irmc.management module

iRMC Management Driver

class ironic.drivers.modules.irmc.management.**IRMCManagement**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.ipmitool.IPMIManagement*

get_properties()

Return the properties of the interface.

Returns Dictionary of <property name>:<property description> entries.

get_secure_boot_state(*task*)

Get the current secure boot state for the node.

NOTE: Not all drivers support this method. Older hardware may not implement that.

Parameters **task** A task from TaskManager.

Raises MissingParameterValue if a required parameter is missing

Raises DriverOperationError or its derivative in case of driver runtime error.

Raises UnsupportedDriverExtension if secure boot is not supported by the driver or the hardware

Returns Boolean

get_sensors_data(*task*)

Get sensors data method.

It gets sensor data from the tasks node via SCCI, and convert the data from XML to the dict format.

Parameters **task** A TaskManager instance.

Raises FailedToGetSensorData when getting the sensor data fails.

Raises FailedToParseSensorData when parsing sensor data fails.

Raises InvalidParameterValue if required parameters are invalid.

Raises MissingParameterValue if a required parameter is missing.

Returns

Returns a consistent formatted dict of sensor data grouped by sensor type, which can be processed by Ceilometer. Example:

```
{
  'Sensor Type 1': {
    'Sensor ID 1': {
      'Sensor Reading': 'Value1 Units1',
      'Sensor ID': 'Sensor ID 1',
      'Units': 'Units1'
    },
    'Sensor ID 2': {
      'Sensor Reading': 'Value2 Units2',
      'Sensor ID': 'Sensor ID 2',
      'Units': 'Units2'
    }
  },
  'Sensor Type 2': {
    'Sensor ID 3': {
      'Sensor Reading': 'Value3 Units3',
      'Sensor ID': 'Sensor ID 3',
      'Units': 'Units3'
    },
    'Sensor ID 4': {
      'Sensor Reading': 'Value4 Units4',
      'Sensor ID': 'Sensor ID 4',
      'Units': 'Units4'
    }
  }
}
```

inject_nmi(*task*)

Inject NMI, Non Maskable Interrupt.

Inject NMI (Non Maskable Interrupt) for a node immediately.

Parameters **task** A TaskManager instance containing the node to act on.

Raises IRMCOperationError on an error from SCCI

Returns None

restore_irmc_bios_config(*task*)

Restore BIOS config for a node.

Parameters **task** a task from TaskManager.

Raises NodeCleaningFailure, on failure to execute step.

Returns None.

`set_boot_device`(*task*, *device*, *persistent=False*)
Set the boot device for a node.

Set the boot device to use on next reboot of the node.

Parameters

- **task** A task from TaskManager.
- **device** The boot device, one of the supported devices listed in *ironic.common.boot_devices*.
- **persistent** Boolean value. True if the boot device will persist to all future boots, False if not. Default: False.

Raises InvalidParameterValue if an invalid boot device is specified.

Raises MissingParameterValue if a required parameter is missing.

Raises IPMIFailure on an error from ipmitool.

`set_secure_boot_state`(*task*, *state*)
Set the current secure boot state for the node.

NOTE: Not all drivers support this method. Older hardware may not implement that.

Parameters

- **task** A task from TaskManager.
- **state** A new state as a boolean.

Raises MissingParameterValue if a required parameter is missing

Raises DriverOperationError or its derivative in case of driver runtime error.

Raises UnsupportedDriverExtension if secure boot is not supported by the driver or the hardware

`validate`(*task*)
Validate the driver-specific management information.

This method validates whether the driver_info property of the supplied node contains the required information for this driver.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue if required parameters are invalid.

Raises MissingParameterValue if a required parameter is missing.

`ironic.drivers.modules.irmc.management.backup_bios_config`(*task*)
Backup BIOS config from a node.

Parameters **task** a TaskManager instance containing the node to act on.

Raises IRMCOperationError on failure.

ironic.drivers.modules.irmc.packaging_version module

exception ironic.drivers.modules.irmc.packaging_version.**InvalidVersion**
Bases: ValueError

Raised when a version string is not a valid version.

```
>>> Version("invalid")
Traceback (most recent call last):
    ...
packaging.version.InvalidVersion: Invalid version: 'invalid'
```

ironic.drivers.modules.irmc.packaging_version.**VERSION_PATTERN**
A string containing the regular expression used to match a valid version.

The pattern is not anchored at either end, and is intended for embedding in larger expressions (for example, matching a version number as part of a file name). The regular expression should be compiled with the re.VERBOSE and re.IGNORECASE flags set.

class ironic.drivers.modules.irmc.packaging_version.**Version**(*version: str*)
Bases: ironic.drivers.modules.irmc.packaging_version._BaseVersion

This class abstracts handling of a projects versions.

A *Version* instance is comparison aware and can be compared and sorted using the standard Python interfaces.

```
>>> v1 = Version("1.0a5")
>>> v2 = Version("1.0")
>>> v1
<Version('1.0a5')>
>>> v2
<Version('1.0')>
>>> v1 < v2
True
>>> v1 == v2
False
>>> v1 > v2
False
>>> v1 >= v2
False
>>> v1 <= v2
True
```

property base_version: str
The base version of the version.

```
>>> Version("1.2.3").base_version
'1.2.3'
>>> Version("1.2.3+abc").base_version
'1.2.3'
```

(continues on next page)

(continued from previous page)

```
>>> Version("1!1.2.3+abc.dev1").base_version
'1!1.2.3'
```

The base version is the public version of the project without any pre or post release markers.

property dev: Optional[int]

The development number of the version.

```
>>> print(Version("1.2.3").dev)
None
>>> Version("1.2.3.dev1").dev
1
```

property epoch: int

The epoch of the version.

```
>>> Version("2.0.0").epoch
0
>>> Version("1!2.0.0").epoch
1
```

property is_devrelease: bool

Whether this version is a development release.

```
>>> Version("1.2.3").is_devrelease
False
>>> Version("1.2.3.dev1").is_devrelease
True
```

property is_postrelease: bool

Whether this version is a post-release.

```
>>> Version("1.2.3").is_postrelease
False
>>> Version("1.2.3.post1").is_postrelease
True
```

property is_prerelease: bool

Whether this version is a pre-release.

```
>>> Version("1.2.3").is_prerelease
False
>>> Version("1.2.3a1").is_prerelease
True
>>> Version("1.2.3b1").is_prerelease
True
>>> Version("1.2.3rc1").is_prerelease
True
>>> Version("1.2.3dev1").is_prerelease
True
```

property local: Optional[str]
The local version segment of the version.

```
>>> print(Version("1.2.3").local)
None
>>> Version("1.2.3+abc").local
'abc'
```

property major: int
The first item of *release* or `0` if unavailable.

```
>>> Version("1.2.3").major
1
```

property micro: int
The third item of *release* or `0` if unavailable.

```
>>> Version("1.2.3").micro
3
>>> Version("1").micro
0
```

property minor: int
The second item of *release* or `0` if unavailable.

```
>>> Version("1.2.3").minor
2
>>> Version("1").minor
0
```

property post: Optional[int]
The post-release number of the version.

```
>>> print(Version("1.2.3").post)
None
>>> Version("1.2.3.post1").post
1
```

property pre: Optional[Tuple[str, int]]
The pre-release segment of the version.

```
>>> print(Version("1.2.3").pre)
None
>>> Version("1.2.3a1").pre
('a', 1)
>>> Version("1.2.3b1").pre
('b', 1)
>>> Version("1.2.3rc1").pre
('rc', 1)
```

property public: str
The public portion of the version.

```
>>> Version("1.2.3").public
'1.2.3'
>>> Version("1.2.3+abc").public
'1.2.3'
>>> Version("1.2.3+abc.dev1").public
'1.2.3'
```

property release: Tuple[int, ...]

The components of the release segment of the version.

```
>>> Version("1.2.3").release
(1, 2, 3)
>>> Version("2.0.0").release
(2, 0, 0)
>>> Version("1!2.0.0.post0").release
(2, 0, 0)
```

Includes trailing zeroes but not the epoch or any pre-release / development / post-release suffixes.

ironic.drivers.modules.irmc.packaging_version.**parse**(*version: str*) → *ironic.drivers.modules.irmc.packaging_version.Vers*

Parse the given version string.

```
>>> parse('1.0.dev1')
<Version('1.0.dev1')>
```

Parameters **version** The version string to parse.

Raises *InvalidVersion* When the version string is not a valid version.

ironic.drivers.modules.irmc.power module

iRMC Power Driver using the Base Server Profile

class ironic.drivers.modules.irmc.power.**IRMCPower**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.PowerInterface*

Interface for power-related actions.

get_power_state(*task*)

Return the power state of the tasks node.

Parameters **task** a TaskManager instance containing the node to act on.

Returns a power state. One of *ironic.common.states*.

Raises InvalidParameterValue if required ipmi parameters are missing.

Raises MissingParameterValue if a required parameter is missing.

Raises IPMIFailure on an error from ipmitool (from _power_status call).

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

get_supported_power_states(*task*)
Get a list of the supported power states.

Parameters **task** A TaskManager instance containing the node to act on. currently not used.

Returns A list with the supported power states defined in *ironic.common.states*.

reboot(*task*, *timeout=None*)
Perform a hard reboot of the tasks node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **timeout** timeout (in seconds) positive integer (> 0) for any power state. None indicates default timeout.

Raises InvalidParameterValue if an invalid power state was specified.

Raises IRMCOperationError if failed to set the power state.

set_power_state(*task*, *power_state*, *timeout=None*)
Set the power state of the tasks node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **power_state** Any power state from *ironic.common.states*.
- **timeout** timeout (in seconds) positive integer (> 0) for any power state. None indicates default timeout.

Raises InvalidParameterValue if an invalid power state was specified.

Raises MissingParameterValue if some mandatory information is missing on the node

Raises IRMCOperationError if failed to set the power state.

validate(*task*)
Validate the driver-specific Node power info.

This method validates whether the driver_info property of the supplied node contains the required information for this driver to manage the power state of the node.

Parameters **task** a TaskManager instance containing the node to act on.

Raises InvalidParameterValue if required driver_info attribute is missing or invalid on the node.

Raises MissingParameterValue if a required parameter is missing.

ironic.drivers.modules.irmc.power.**METRICS** =
<ironic_lib.metrics.NoopMetricLogger object>
SC2.mib: sc2srvCurrentBootStatus returns status of the current boot

ironic.drivers.modules.irmc.raid module

Irmc RAID specific methods

class ironic.drivers.modules.irmc.raid.**IRMCRAID**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.RAIDInterface*

create_configuration(*task*, *create_root_volume=True*, *create_nonroot_volumes=True*)
Create the RAID configuration.

This method creates the RAID configuration on the given node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **create_root_volume** If True, a root volume is created during RAID configuration. Otherwise, no root volume is created. Default is True.
- **create_nonroot_volumes** If True, non-root volumes are created. If False, no non-root volumes are created. Default is True.

Returns states.CLEANWAIT if RAID configuration is in progress asynchronously.

Raises MissingParameterValue, if node.target_raid_config is missing or empty.

Raises IRMCOperationError on an error from scciclient

delete_configuration(*task*)
Delete the RAID configuration.

Parameters **task** a TaskManager instance containing the node to act on.

Returns states.CLEANWAIT if deletion is in progress asynchronously or None if it is complete.

get_properties()
Return the properties of the interface.

Module contents

ironic.drivers.modules.network package

Submodules

ironic.drivers.modules.network.common module

class ironic.drivers.modules.network.common.**NeutronVIFPortIDMixin**
Bases: *ironic.drivers.modules.network.common.VIFPortIDMixin*

VIF port ID mixin class for neutron network interfaces.

Mixin class that provides VIF-related network interface methods for neutron network interfaces. On VIF attach/detach, the associated neutron port will be updated.

get_node_network_data(*task*)
Get network configuration data for node ports.

Pull network data from ironic node object if present, otherwise collect it for Neutron VIFs.

Parameters **task** A TaskManager instance.

Raises InvalidParameterValue, if the network interface configuration is invalid.

Raises MissingParameterValue, if some parameters are missing.

Returns a dict holding network configuration information adhearing Nova network metadata layout (*network_data.json*).

port_changed(*task*, *port_obj*)
Handle any actions required when a port changes

Parameters

- **task** a TaskManager instance.
- **port_obj** a changed Port object from the API before it is saved to database.

Raises FailedToUpdateDHCPOptOnPort, Conflict

portgroup_changed(*task*, *portgroup_obj*)
Handle any actions required when a portgroup changes

Parameters

- **task** a TaskManager instance.
- **portgroup_obj** a changed Portgroup object from the API before it is saved to database.

Raises FailedToUpdateDHCPOptOnPort, Conflict

vif_attach(*task*, *vif_info*)
Attach a virtual network interface to a node

Attach a virtual interface to a node. When selecting a port or portgroup to attach the virtual interface to, the following ordered criteria are applied:

- Require ports or portgroups to have a physical network that is either None or one of the VIFs allowed physical networks.
- Prefer ports or portgroups with a physical network field which is not None.
- Prefer portgroups to ports.
- Prefer ports with PXE enabled.

Parameters

- **task** A TaskManager instance.
- **vif_info** a dictionary of information about a VIF. It must have an id key, whose value is a unique identifier for that VIF.

Raises NetworkError, VifAlreadyAttached, NoFreePhysicalPorts

Raises PortgroupPhysnetInconsistent if one of the nodes portgroups has ports which are not all assigned the same physical network.

vif_detach(*task*, *vif_id*)
Detach a virtual network interface from a node

Parameters

- **task** A TaskManager instance.
- **vif_id** A VIF ID to detach

Raises VifNotAttached if VIF not attached.

Raises NetworkError if unbind Neutron port failed.

class ironic.drivers.modules.network.common.**VIFPortIDMixin**
Bases: object

VIF port ID mixin class for non-neutron network interfaces.

Mixin class that provides VIF-related network interface methods for non-neutron network interfaces. There are no effects due to VIF attach/detach that are external to ironic.

NOTE: This does not yet support the full set of VIF methods, as it does not provide vif_attach, vif_detach, port_changed, or portgroup_changed.

get_current_vif(*task*, *p_obj*)
Returns the currently used VIF associated with port or portgroup

We are booting the node only in one network at a time, and presence of cleaning_vif_port_id means were doing cleaning, of provisioning_vif_port_id - provisioning, of rescuing_vif_port_id - rescuing. Otherwise its a tenant network

Parameters

- **task** A TaskManager instance.
- **p_obj** Ironic port or portgroup object.

Returns VIF ID associated with p_obj or None.

vif_list(*task*)
List attached VIF IDs for a node

Parameters **task** A TaskManager instance.

Returns List of VIF dictionaries, each dictionary will have an id entry with the ID of the VIF.

ironic.drivers.modules.network.common.**get_free_port_like_object**(*task*, *vif_id*, *physnets*, *vif_info={}*)
Find free port-like object (portgroup or port) VIF will be attached to.

Ensures that the VIF is not already attached to this node. When selecting a port or portgroup to attach the virtual interface to, the following ordered criteria are applied:

- Require ports or portgroups to have a physical network that is either None or one of the VIFs allowed physical networks.
- Prefer ports or portgroups with a physical network field which is not None.
- Prefer portgroups to ports.
- Prefer ports with PXE enabled.

Parameters

- **task** a TaskManager instance.
- **vif_id** Name or UUID of a VIF.
- **physnets** Set of physical networks on which the VIF may be attached. This is governed by the segments of the VIFs network. An empty set indicates that the ports physical networks should be ignored.
- **vif_info** dict that may contain extra information, such as port_uuid

Raises VifAlreadyAttached, if VIF is already attached to the node.

Raises NoFreePhysicalPorts, if there is no port-like object VIF can be attached to.

Raises PortgroupPhysnetInconsistent if one of the nodes portgroups has ports which are not all assigned the same physical network.

Returns port-like object VIF will be attached to.

ironic.drivers.modules.network.common.**plug_port_to_tenant_network**(*task*, *port_like_obj*, *client=None*)

Plug port like object to tenant network.

Parameters

- **task** A TaskManager instance.
- **port_like_obj** port-like object to plug.
- **client** Neutron client instance.

Raises NetworkError if failed to update Neutron port.

Raises VifNotAttached if tenant VIF is not associated with port_like_obj.

ironic.drivers.modules.network.flat module

Flat network interface. Useful for shared, flat networks.

class ironic.drivers.modules.network.flat.**FlatNetwork**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.network.common.NeutronVIFPortIDMixin*, *ironic.common.neutron.NeutronNetworkInterfaceMixin*, *ironic.drivers.base.NetworkInterface*

Flat network interface.

add_cleaning_network(*task*)

Add the cleaning network to a node.

Parameters **task** A TaskManager instance.

Returns a dictionary in the form {port.uuid: neutron_port[id]}

Raises NetworkError, InvalidParameterValue

add_inspection_network(*task*)

Add the inspection network to the node.

Parameters task A TaskManager instance.

Returns a dictionary in the form {port.uuid: neutron_port[id]}

Raises NetworkError

Raises InvalidParameterValue, if the network interface configuration is invalid.

add_provisioning_network(*task*)
Add the provisioning network to a node.

Parameters task A TaskManager instance.

Raises NetworkError when failed to set binding:host_id

add_rescuing_network(*task*)
Add the rescuing network to a node.

Flat network does not use the rescuing network. Bind the port again since unconfigure_tenant_network() unbound it.

Parameters task A TaskManager instance.

Returns a dictionary in the form {port.uuid: neutron_port[id]}

Raises NetworkError, InvalidParameterValue

configure_tenant_networks(*task*)
Configure tenant networks for a node.

Parameters task A TaskManager instance.

remove_cleaning_network(*task*)
Remove the cleaning network from a node.

Parameters task A TaskManager instance.

Raises NetworkError

remove_inspection_network(*task*)
Removes the inspection network from a node.

Parameters task A TaskManager instance.

Raises InvalidParameterValue, if the network interface configuration is invalid.

Raises MissingParameterValue, if some parameters are missing.

remove_provisioning_network(*task*)
Remove the provisioning network from a node.

Parameters task A TaskManager instance.

remove_rescuing_network(*task*)
Remove the rescuing network from a node.

Flat network does not use the rescuing network. Unbind the port again since add_rescuing_network() bound it.

Parameters task A TaskManager instance.

Raises NetworkError

unconfigure_tenant_networks(*task*)
Unconfigure tenant networks for a node.

Unbind the port here/now to avoid the possibility of the ironic port being bound to the tenant and cleaning networks at the same time.

Parameters **task** A TaskManager instance.

Raises NetworkError

validate(*task*)
Validates the network interface.

Parameters **task** a TaskManager instance.

Raises InvalidParameterValue, if the network interface configuration is invalid.

Raises MissingParameterValue, if some parameters are missing.

ironic.drivers.modules.network.neutron module

class ironic.drivers.modules.network.neutron.**NeutronNetwork**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.network.common.NeutronVIFPortIDMixin*, *ironic.common.neutron.NeutronNetworkInterfaceMixin*, *ironic.drivers.base.NetworkInterface*

Neutron v2 network interface

add_cleaning_network(*task*)
Create neutron ports for each port on task.node to boot the ramdisk.

Parameters **task** a TaskManager instance.

Raises NetworkError

Returns a dictionary in the form {port.uuid: neutron_port[id]}

add_inspection_network(*task*)
Add the inspection network to the node.

Parameters **task** A TaskManager instance.

Returns a dictionary in the form {port.uuid: neutron_port[id]}

Raises NetworkError

Raises InvalidParameterValue, if the network interface configuration is invalid.

add_provisioning_network(*task*)
Add the provisioning network to a node.

Parameters **task** A TaskManager instance.

Raises NetworkError

add_rescuing_network(*task*)
Create neutron ports for each port to boot the rescue ramdisk.

Parameters **task** a TaskManager instance.

Returns a dictionary in the form {port.uuid: neutron_port[id]}

configure_tenant_networks(*task*)
Configure tenant networks for a node.

Parameters **task** A TaskManager instance.

Raises NetworkError

need_power_on(*task*)
Check if the node has any Smart NIC ports

Parameters **task** A TaskManager instance.

Returns A boolean to indicate Smart NIC port presence

remove_cleaning_network(*task*)
Deletes the neutron port created for booting the ramdisk.

Parameters **task** a TaskManager instance.

Raises NetworkError

remove_inspection_network(*task*)
Removes the inspection network from a node.

Parameters **task** A TaskManager instance.

Raises InvalidParameterValue, if the network interface configuration is invalid.

Raises MissingParameterValue, if some parameters are missing.

remove_provisioning_network(*task*)
Remove the provisioning network from a node.

Parameters **task** A TaskManager instance.

Raises NetworkError

remove_rescuing_network(*task*)
Deletes neutron port created for booting the rescue ramdisk.

Parameters **task** a TaskManager instance.

Raises NetworkError

unconfigure_tenant_networks(*task*)
Unconfigure tenant networks for a node.

Nova takes care of port removal from tenant network, we unbind it here/now to avoid the possibility of the ironic port being bound to the tenant and cleaning networks at the same time.

Parameters **task** A TaskManager instance.

Raises NetworkError

validate(*task*)
Validates the network interface.

Parameters **task** a TaskManager instance.

Raises InvalidParameterValue, if the network interface configuration is invalid.

Raises MissingParameterValue, if some parameters are missing.

validate_rescue(*task*)
Validates the network interface for rescue operation.

Parameters **task** a TaskManager instance.

Raises InvalidParameterValue, if the network interface configuration is invalid.

Raises MissingParameterValue, if some parameters are missing.

ironic.drivers.modules.network.noop module

class ironic.drivers.modules.network.noop.**NoopNetwork**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.NetworkInterface*

Noop network interface.

add_cleaning_network(*task*)
Add the cleaning network to a node.

Parameters **task** A TaskManager instance.

add_provisioning_network(*task*)
Add the provisioning network to a node.

Parameters **task** A TaskManager instance.

configure_tenant_networks(*task*)
Configure tenant networks for a node.

Parameters **task** A TaskManager instance.

get_current_vif(*task*, *p_obj*)
Returns the currently used VIF associated with port or portgroup

We are booting the node only in one network at a time, and presence of cleaning_vif_port_id means were doing cleaning, of provisioning_vif_port_id - provisioning of rescuing_vif_port_id - rescuing. Otherwise its a tenant network

Parameters

- **task** A TaskManager instance.
- **p_obj** Ironic port or portgroup object.

Returns VIF ID associated with p_obj or None.

port_changed(*task*, *port_obj*)
Handle any actions required when a port changes

Parameters

- **task** a TaskManager instance.
- **port_obj** a changed Port object.

Raises Conflict, FailedToUpdateDHCPOptOnPort

portgroup_changed(*task*, *portgroup_obj*)
Handle any actions required when a portgroup changes

Parameters

- **task** a TaskManager instance.
- **portgroup_obj** a changed Portgroup object.

Raises Conflict, FailedToUpdateDHCPOptOnPort

remove_cleaning_network(*task*)
Remove the cleaning network from a node.

Parameters **task** A TaskManager instance.

remove_provisioning_network(*task*)
Remove the provisioning network from a node.

Parameters **task** A TaskManager instance.

unconfigure_tenant_networks(*task*)
Unconfigure tenant networks for a node.

Parameters **task** A TaskManager instance.

validate_inspection(*task*)
Validate that the node has required properties for inspection.

Parameters **task** A TaskManager instance with the node being checked

vif_attach(*task*, *vif_info*)
Attach a virtual network interface to a node

Parameters

- **task** A TaskManager instance.
- **vif_info** a dictionary of information about a VIF. It must have an id key, whose value is a unique identifier for that VIF.

Raises NetworkError, VifAlreadyAttached, NoFreePhysicalPorts

vif_detach(*task*, *vif_id*)
Detach a virtual network interface from a node

Parameters

- **task** A TaskManager instance.
- **vif_id** A VIF ID to detach

Raises NetworkError, VifNotAttached

vif_list(*task*)
List attached VIF IDs for a node.

Parameters **task** A TaskManager instance.

Returns List of VIF dictionaries, each dictionary will have an id entry with the ID of the VIF.

Module contents

ironic.drivers.modules.redfish package

Submodules

ironic.drivers.modules.redfish.bios module

class ironic.drivers.modules.redfish.bios.**RedfishBIOS**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.BIOSInterface*

apply_configuration(*task*, *settings*)
Apply the BIOS settings to the node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **settings** a list of BIOS settings to be updated.

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError on an error from the Sushy library

cache_bios_settings(*task*)
Store or update the current BIOS settings for the node.

Get the current BIOS settings and store them in the bios_settings database table.

Parameters **task** a TaskManager instance containing the node to act on.

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError on an error from the Sushy library

Raises UnsupportedDriverExtension if the system does not support BIOS settings

factory_reset(*task*)
Reset the BIOS settings of the node to the factory default.

Parameters **task** a TaskManager instance containing the node to act on.

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError on an error from the Sushy library

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

post_configuration(*task*, *settings*)
Perform post configuration action to store the BIOS settings.

Extension point to allow vendor implementations to extend this class and override this method to perform a custom action to write the BIOS settings to the Redfish service. The default implementation performs a reboot.

Parameters

- **task** a TaskManager instance containing the node to act on.

- **settings** a list of BIOS settings to be updated.

post_reset(*task*)
Perform post reset action to apply the BIOS factory reset.

Extension point to allow vendor implementations to extend this class and override this method to perform a custom action to apply the BIOS factory reset to the Redfish service. The default implementation performs a reboot.

Parameters **task** a TaskManager instance containing the node to act on.

validate(*task*)
Validates the driver information needed by the redfish driver.

Parameters **task** a TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

ironic.drivers.modules.redfish.boot module

class ironic.drivers.modules.redfish.boot.**RedfishVirtualMediaBoot**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.BootInterface*

Virtual media boot interface over Redfish.

Virtual Media allows booting the system from the virtual CD/DVD drive containing the user image that BMC inserts into the drive.

The CD/DVD images must be in ISO format and (depending on BMC implementation) could be pulled over HTTP, served as iSCSI targets or NFS volumes.

The baseline boot workflow looks like this:

1. Pull kernel, ramdisk and ESP (FAT partition image with EFI boot loader) images (ESP is only needed for UEFI boot)
2. Create bootable ISO out of images (#1), push it to Glance and pass to the BMC as Swift temporary URL
3. Optionally create floppy image with desired system configuration data, push it to Glance and pass to the BMC as Swift temporary URL
4. Insert CD/DVD and (optionally) floppy images and set proper boot mode

For building deploy or rescue ISO, redfish boot interface uses *deploy_kernel/deploy_ramdisk* or *rescue_kernel/rescue_ramdisk* properties from *[instance_info]* or *[driver_info]*.

For building boot (user) ISO, redfish boot interface seeks *kernel_id* and *ramdisk_id* properties in the Glance image metadata found in *[instance_info]image_source* node property.

capabilities = ['iscsi_volume_boot', 'ramdisk_boot', 'ramdisk_boot_configdrive']

clean_up_instance(*task*)
Cleans up the boot of instance.

This method cleans up the environment that was setup for booting the instance.

Parameters `task` A task from TaskManager.

Returns None

`clean_up_ramdisk`(*task*)
Cleans up the boot of ironic ramdisk.

This method cleans up the environment that was setup for booting the deploy ramdisk.

Parameters `task` A task from TaskManager.

Returns None

`get_properties`()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

`prepare_instance`(*task*)
Prepares the boot of instance over virtual media.

This method prepares the boot of the instance after reading relevant information from the nodes instance_info.

The internal logic is as follows:

- If *boot_option* requested for this deploy is local, then set the node to boot from disk.
- Unless *boot_option* requested for this deploy is ramdisk, pass root disk/partition ID to virtual media boot image
- Otherwise build boot image, insert it into virtual media device and set node to boot from CD.

Parameters `task` a task from TaskManager.

Returns None

Raises InstanceDeployFailure, if its try to boot iSCSI volume in BIOS boot mode.

`prepare_ramdisk`(*task*, *ramdisk_params*)
Prepares the boot of deploy or rescue ramdisk over virtual media.

This method prepares the boot of the deploy or rescue ramdisk after reading relevant information from the nodes driver_info and instance_info.

Parameters

- `task` A task from TaskManager.
- `ramdisk_params` the parameters to be passed to the ramdisk.

Returns None

Raises MissingParameterValue, if some information is missing in nodes driver_info or instance_info.

Raises InvalidParameterValue, if some information provided is invalid.

Raises IronicException, if some power or set boot boot device operation failed on the node.

validate(*task*)
Validate the deployment information for the tasks node.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

validate_inspection(*task*)
Validate that the node has required properties for inspection.

Parameters **task** A TaskManager instance with the node being checked

Raises MissingParameterValue if node is missing one or more required parameters

Raises UnsupportedDriverExtension

ironic.drivers.modules.redfish.boot.**eject_vmedia**(*task*, *boot_device=None*)
Eject virtual CDs and DVDs

Parameters

- **task** A task from TaskManager.
- **boot_device** sushy boot device e.g. *VIRTUAL_MEDIA_CD*, *VIRTUAL_MEDIA_DVD* or *VIRTUAL_MEDIA_FLOPPY* or *None* to eject everything (default).

Raises InvalidParameterValue, if no suitable virtual CD or DVD is found on the node.

ironic.drivers.modules.redfish.firmware_utils module

ironic.drivers.modules.redfish.firmware_utils.**cleanup**(*node*)
Clean up staged files

Parameters **node** Node for which to clean up. Should contain firmware_cleanup entry in *driver_internal_info* to indicate source(s) to be cleaned up.

ironic.drivers.modules.redfish.firmware_utils.**download_to_temp**(*node*, *url*)
Downloads to temporary location from given URL

Parameters

- **node** Node for which to download to temporary location
- **url** URL to download from

Returns File path of temporary location file is downloaded to

ironic.drivers.modules.redfish.firmware_utils.**get_swift_temp_url**(*parsed_url*)
Gets Swift temporary URL

Parameters **parsed_url** Parsed URL from URL in format swift://container/[sub-folder/]file

Returns Swift temporary URL

ironic.drivers.modules.redfish.firmware_utils.**stage**(*node*, *source*, *temp_file*)
Stage temporary file to configured location

Parameters

- **node** Node for which to stage the file
- **source** Where to stage the file. Corresponds to CONF.redfish.firmware_source.
- **temp_file** File path of temporary file to stage

Returns Tuple of staged URL and source (http or swift) that needs cleanup of staged files afterwards.

Raises *RedfishError* If staging to HTTP server has failed.

ironic.drivers.modules.redfish.firmware_utils.**validate_update_firmware_args**(*firmware_images*)
Validate `update_firmware` step input argument

Parameters **firmware_images** args to validate.

Raises InvalidParameterValue When argument is not valid

ironic.drivers.modules.redfish.firmware_utils.**verify_checksum**(*node*, *checksum*, *file_path*)
Verify checksum.

Parameters

- **node** Node for which file to verify checksum
- **checksum** Expected checksum value
- **file_path** File path for which to verify checksum

Raises *RedfishError* When checksum does not match

ironic.drivers.modules.redfish.inspect module

Redfish Inspect Interface

class ironic.drivers.modules.redfish.inspect.**RedfishInspect**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.InspectInterface*

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

inspect_hardware(*task*)
Inspect hardware to get the hardware properties.

Inspects hardware to get the essential properties. It fails if any of the essential properties are not received from the node.

Parameters **task** a TaskManager instance.

Raises HardwareInspectionFailure if essential properties could not be retrieved successfully.

Returns The resulting state of inspection.

validate(*task*)
Validate the driver-specific Node deployment info.

This method validates whether the driver_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

ironic.drivers.modules.redfish.management module

class ironic.drivers.modules.redfish.management.**RedfishManagement**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.ManagementInterface*

clear_secure_boot_keys(*task*)
Clear all secure boot keys.

Parameters **task** a task from TaskManager.

Raises UnsupportedDriverExtension if secure boot is now supported.

Raises RedfishError on runtime driver error.

detect_vendor(*task*)
Detects and returns the hardware vendor.

Uses the Systems Manufacturer field.

Parameters **task** A task from TaskManager.

Raises InvalidParameterValue if an invalid component, indicator or state is specified.

Raises MissingParameterValue if a required parameter is missing

Raises RedfishError on driver-specific problems.

Returns String representing the BMC reported Vendor or Manufacturer, otherwise returns None.

get_boot_device(*task*)
Get the current boot device for a node.

Parameters **task** a task from TaskManager.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError on an error from the Sushy library

Returns

a dictionary containing:

boot_device the boot device, one of *ironic.common.boot_devices* or None if it is unknown.

persistent Boolean value or None, True if the boot device persists, False otherwise. None if its unknown.

`get_boot_mode`(*task*)

Get the current boot mode for a node.

Provides the current boot mode of the node.

Parameters **task** A task from TaskManager.

Raises MissingParameterValue if a required parameter is missing

Raises DriverOperationError or its derivative in case of driver runtime error.

Returns The boot mode, one of `ironic.common.boot_mode` or None if it is unknown.

`get_indicator_state`(*task*, *component*, *indicator*)

Get current state of the indicator of the hardware component.

Parameters

- **task** A task from TaskManager.
- **component** The hardware component, one of *ironic.common.components*.
- **indicator** Indicator ID (as reported by *get_supported_indicators*).

Raises MissingParameterValue if a required parameter is missing

Raises RedfishError on an error from the Sushy library

Returns Current state of the indicator, one of *ironic.common.indicator_states*.

`get_mac_addresses`(*task*)

Get MAC address information for the node.

Parameters **task** A TaskManager instance containing the node to act on.

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError on an error from the Sushy library

Returns A list of MAC addresses for the node

`get_properties`()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

`get_secure_boot_state`(*task*)

Get the current secure boot state for the node.

Parameters **task** A task from TaskManager.

Raises MissingParameterValue if a required parameter is missing

Raises RedfishError or its derivative in case of a driver runtime error.

Raises UnsupportedDriverExtension if secure boot is not supported by the hardware.

Returns Boolean

`get_sensors_data`(*task*)
Get sensors data.

Parameters **task** a TaskManager instance.

Raises FailedToGetSensorData when getting the sensor data fails.

Raises FailedToParseSensorData when parsing sensor data fails.

Raises InvalidParameterValue if required parameters are missing.

Raises MissingParameterValue if a required parameter is missing.

Returns returns a dict of sensor data grouped by sensor type.

`get_supported_boot_devices`(*task*)
Get a list of the supported boot devices.

Parameters **task** a task from TaskManager.

Returns A list with the supported boot devices defined in *ironic.common.boot_devices*.

`get_supported_boot_modes`(*task*)
Get a list of the supported boot modes.

Parameters **task** A task from TaskManager.

Returns A list with the supported boot modes defined in *ironic.common.boot_modes*. If boot mode support cant be determined, empty list is returned.

`get_supported_indicators`(*task*, *component=None*)
Get a map of the supported indicators (e.g. LEDs).

Parameters

- **task** A task from TaskManager.
- **component** If not *None*, return indicator information for just this component, otherwise return indicators for all existing components.

Returns

A dictionary of hardware components (*ironic.common.components*) as keys with values being dictionaries having indicator IDs as keys and indicator properties as values.

```
{
    'chassis': {
        'enclosure-0': {
            "readonly": true,
            "states": [
                "OFF",
                "ON"
```

(continues on next page)

(continued from previous page)

```
            ]
        }
    },
    'system':
        'blade-A': {
            "readonly": true,
            "states": [
                "OFF",
                "ON"
            ]
        }
    },
    'drive':
        'ssd0': {
            "readonly": true,
            "states": [
                "OFF",
                "ON"
            ]
        }
    }
}
```

inject_nmi(*task*)

Inject NMI, Non Maskable Interrupt.

Inject NMI (Non Maskable Interrupt) for a node immediately.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError on an error from the Sushy library

reset_secure_boot_keys_to_default(*task*)

Reset secure boot keys to manufacturing defaults.

Parameters **task** a task from TaskManager.

Raises UnsupportedDriverExtension if secure boot is now supported.

Raises RedfishError on runtime driver error.

restore_boot_device(*task*, *system*)

Restore boot device if needed.

Checks the redfish_boot_device internal flag and sets the one-time boot device accordingly. A warning is issued if it fails.

This method is supposed to be called from the Redfish power interface and should be considered private to the Redfish hardware type.

Parameters

- **task** a task from TaskManager.
- **system** a Redfish System object.

set_boot_device(*task*, *device*, *persistent=False*)
Set the boot device for a node.

Set the boot device to use on next reboot of the node.

Parameters

- **task** a task from TaskManager.
- **device** the boot device, one of *ironic.common.boot_devices*.
- **persistent** Boolean value. True if the boot device will persist to all future boots, False if not. Default: False.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError on an error from the Sushy library

set_boot_mode(*task*, *mode*)
Set the boot mode for a node.

Set the boot mode to use on next reboot of the node.

Parameters

- **task** A task from TaskManager.
- **mode** The boot mode, one of *ironic.common.boot_modes*.

Raises InvalidParameterValue if an invalid boot mode is specified.

Raises MissingParameterValue if a required parameter is missing

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError on an error from the Sushy library

set_indicator_state(*task*, *component*, *indicator*, *state*)
Set indicator on the hardware component to the desired state.

Parameters

- **task** A task from TaskManager.
- **component** The hardware component, one of *ironic.common.components*.
- **indicator** Indicator ID (as reported by *get_supported_indicators*).
- **state** Desired state of the indicator, one of *ironic.common.indicator_states*.

Raises InvalidParameterValue if an invalid component, indicator or state is specified.

Raises MissingParameterValue if a required parameter is missing

Raises RedfishError on an error from the Sushy library

set_secure_boot_state(*task*, *state*)
Set the current secure boot state for the node.

Parameters

- **task** A task from TaskManager.
- **state** A new state as a boolean.

Raises MissingParameterValue if a required parameter is missing

Raises RedfishError or its derivative in case of a driver runtime error.

Raises UnsupportedDriverExtension if secure boot is not supported by the hardware.

update_firmware(*task*, *firmware_images*)
Updates the firmware on the node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **firmware_images** A list of firmware images are to apply.

Returns None if it is completed.

Raises RedfishError on an error from the Sushy library.

validate(*task*)
Validates the driver information needed by the redfish driver.

Parameters **task** a TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

ironic.drivers.modules.redfish.power module

class ironic.drivers.modules.redfish.power.**RedfishPower**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.PowerInterface*

get_power_state(*task*)
Get the current power state of the tasks node.

Parameters **task** a TaskManager instance containing the node to act on.

Returns a power state. One of *ironic.common.states*.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError on an error from the Sushy library

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

get_supported_power_states(*task*)
Get a list of the supported power states.

Parameters **task** A TaskManager instance containing the node to act on. Not used by this driver at the moment.

Returns A list with the supported power states defined in *ironic.common.states*.

reboot(*task*, *timeout=None*)
Perform a hard reboot of the tasks node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **timeout** Time to wait for the node to become powered on.

Raises MissingParameterValue if a required parameter is missing.

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError on an error from the Sushy library

set_power_state(*task*, *power_state*, *timeout=None*)
Set the power state of the tasks node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **power_state** Any power state from *ironic.common.states*.
- **timeout** Time to wait for the node to reach the requested state.

Raises MissingParameterValue if a required parameter is missing.

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError on an error from the Sushy library

validate(*task*)
Validates the driver information needed by the redfish driver.

Parameters **task** a TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

ironic.drivers.modules.redfish.raid module

class ironic.drivers.modules.redfish.raid.**RedfishRAID**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.RAIDInterface*

apply_configuration(*task*, *raid_config*, *create_root_volume=True*, *create_nonroot_volumes=False*, *delete_existing=False*)
Applies RAID configuration on the given node.

Parameters

- **task** A TaskManager instance.

- **raid_config** The RAID configuration to apply.
- **create_root_volume** Setting this to False indicates not to create root volume that is specified in raid_config. Default value is True.
- **create_nonroot_volumes** Setting this to False indicates not to create non-root volumes (all except the root volume) in raid_config. Default value is True.
- **delete_existing** Setting this to True indicates to delete RAID configuration prior to creating the new configuration.

Raises InvalidParameterValue, if the RAID configuration is invalid.

Returns states.DEPLOYWAIT if RAID configuration is in progress asynchronously or None if it is complete.

create_configuration(*task*, *create_root_volume=True*, *create_nonroot_volumes=True*, *delete_existing=False*)

Create RAID configuration on the node.

This method creates the RAID configuration as read from node.target_raid_config. This method by default will create all logical disks.

Parameters

- **task** TaskManager object containing the node.
- **create_root_volume** Setting this to False indicates not to create root volume that is specified in the nodes target_raid_config. Default value is True.
- **create_nonroot_volumes** Setting this to False indicates not to create non-root volumes (all except the root volume) in the nodes target_raid_config. Default value is True.
- **delete_existing** Setting this to True indicates to delete RAID configuration prior to creating the new configuration. Default is False.

Returns states.CLEANWAIT if RAID configuration is in progress asynchronously or None if it is complete.

Raises RedfishError if there is an error creating the configuration

delete_configuration(*task*)

Delete RAID configuration on the node.

Parameters **task** TaskManager object containing the node.

Returns states.CLEANWAIT (cleaning) or states.DEPLOYWAIT (deployment) if deletion is in progress asynchronously or None if it is complete.

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

post_create_configuration(*task*, *raid_configs*, *return_state=None*)

Perform post create_configuration action to commit the config.

Extension point to allow vendor implementations to extend this class and override this method to perform a custom action to commit the RAID create configuration to the Redfish service.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **raid_configs** a list of dictionaries containing the RAID configuration operation details.
- **return_state** state to return based on operation being invoked

post_delete_configuration(*task*, *raid_configs*, *return_state=None*)
Perform post delete_configuration action to commit the config.

Extension point to allow vendor implementations to extend this class and override this method to perform a custom action to commit the RAID delete configuration to the Redfish service.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **raid_configs** a list of dictionaries containing the RAID configuration operation details.
- **return_state** state to return based on operation being invoked

pre_create_configuration(*task*, *logical_disks_to_create*)
Perform required actions before creating config.

Extension point to allow vendor implementations to extend this class and override this method to perform custom actions prior to creating the RAID configuration on the Redfish service.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **logical_disks_to_create** list of logical disks to create.

Returns updated list of logical disks to create.

pre_delete_configuration(*task*, *vols_to_delete*)
Perform required actions before deleting config.

Extension point to allow vendor implementations to extend this class and override this method to perform custom actions prior to deleting the RAID configuration on the Redfish service.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **vols_to_delete** list of volumes to delete.

validate_raid_config(*task*, *raid_config*)
Validates the given RAID configuration.

Parameters

- **task** A TaskManager instance.
- **raid_config** The RAID configuration to validate.

Raises InvalidParameterValue, if the RAID configuration is invalid.

volume_create_error_handler(*task*, *exc*, *volume_collection*, *payload*)
Handle error from failed VolumeCollection.create()

Extension point to allow vendor implementations to extend this class and override this method to perform a custom action if the call to VolumeCollection.create() fails.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **exc** the exception raised by VolumeCollection.create().
- **volume_collection** the sushy VolumeCollection instance.
- **payload** the payload passed to the failed create().

Returns Newly created Volume resource or TaskMonitor if async task.

Raises RedfishError if there is an error creating the virtual disk.

ironic.drivers.modules.redfish.raid.**convert_drive_units**(*logical_disks*, *node*)

Convert size in logical_disks from gb to bytes

ironic.drivers.modules.redfish.raid.**create_virtual_disk**(*task*, *raid_controller*, *physical_disks*, *raid_level*, *size_bytes*, *disk_name=None*, *span_length=None*, *span_depth=None*, *error_handler=None*)

Create a single virtual disk on a RAID controller.

Parameters

- **task** TaskManager object containing the node.
- **raid_controller** id of the RAID controller.
- **physical_disks** ids of the physical disks.
- **raid_level** RAID level of the virtual disk.
- **size_bytes** size of the virtual disk.
- **disk_name** name of the virtual disk. (optional)
- **span_depth** Number of spans in virtual disk. (optional)
- **span_length** Number of disks per span. (optional)
- **error_handler** function to call if volume create fails. (optional)

Returns Newly created Volume resource or TaskMonitor if async task.

Raises RedfishConnectionError when it fails to connect to Redfish.

Raises RedfishError if there is an error creating the virtual disk.

ironic.drivers.modules.redfish.raid.**get_physical_disks**(*node*)

Get the physical drives of the node for RAID controllers.

Parameters **node** an ironic node object.

Returns a list of Drive objects from sushy

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError if there is an error getting the drives via Redfish

ironic.drivers.modules.redfish.raid.**update_raid_config**(*node*)
Updates nodes raid_config field with current logical disks.

Parameters **node** node for which to update the raid_config field

ironic.drivers.modules.redfish.utils module

class ironic.drivers.modules.redfish.utils.**SessionCache**(*driver_info*)
Bases: object

Cache of HTTP sessions credentials

AUTH_CLASSES = {'auto': <class 'sushy.auth.SessionOrBasicAuth'>, 'basic': <class 'sushy.auth.BasicAuth'>, 'session': <class 'sushy.auth.SessionAuth'>}

ironic.drivers.modules.redfish.utils.**get_enabled_macs**(*task*, *system*)
Get information on MAC addresses of enabled ports using Redfish.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **system** a Redfish System object

Returns a dictionary containing MAC addresses of enabled interfaces in a {mac: <state>} format, where <state> is a sushy constant

ironic.drivers.modules.redfish.utils.**get_event_service**(*node*)
Get a nodes event service.

Parameters **node** an Ironic node object.

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError when the EventService is not registered in Redfish

ironic.drivers.modules.redfish.utils.**get_system**(*node*)
Get a Redfish System that represents a node.

Parameters **node** an Ironic node object

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError if the System is not registered in Redfish

ironic.drivers.modules.redfish.utils.**get_task_monitor**(*node*, *uri*)
Get a TaskMonitor for a node.

Parameters

- **node** an Ironic node object
- **uri** the URI of a TaskMonitor

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError when the TaskMonitor is not available in Redfish

ironic.drivers.modules.redfish.utils.**get_update_service**(*node*)
Get a nodes update service.

Parameters **node** an Ironic node object

Raises RedfishConnectionError when it fails to connect to Redfish

Raises RedfishError when the UpdateService is not registered in Redfish

ironic.drivers.modules.redfish.utils.**parse_driver_info**(*node*)
Parse the information required for Ironic to connect to Redfish.

Parameters **node** an Ironic node object

Returns dictionary of parameters

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

ironic.drivers.modules.redfish.utils.**wait_until_get_system_ready**(*node*)
Wait until Redfish system is ready.

Parameters **node** an Ironic node object

Raises RedfishConnectionError on time out.

ironic.drivers.modules.redfish.vendor module

Vendor Interface for Redfish drivers and its supporting methods.

class ironic.drivers.modules.redfish.vendor.**RedfishVendorPassthru**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.VendorInterface*

Vendor-specific interfaces for Redfish drivers.

create_subscription(*task*, ***kwargs*)
Creates a subscription.

Parameters

- **task** A TaskManager object.
- **kwargs** The arguments sent with vendor passthru.

Raises RedfishError, if any problem occurs when trying to create a subscription.

delete_subscription(*task*, ***kwargs*)
Delete a subscription.

Parameters

- **task** A TaskManager object.
- **kwargs** The arguments sent with vendor passthru.

Raises RedfishError, if any problem occurs when trying to delete the subscription.

eject_vmedia(*task*, ***kwargs*)
Eject a virtual media device.

Parameters

- **task** A TaskManager object.

- **kwargs** The arguments sent with vendor passthru. The optional kwargs are:: boot_device: the boot device to eject

get_all_subscriptions(*task*, ***kwargs*)
Get all Subscriptions on the node

Parameters

- **task** A TaskManager object.
- **kwargs** Not used.

Raises RedfishError, if any problem occurs when retrieving all subscriptions.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

get_subscription(*task*, ***kwargs*)
Get a specific subscription on the node

Parameters

- **task** A TaskManager object.
- **kwargs** The arguments sent with vendor passthru.

Raises RedfishError, if any problem occurs when retrieving the subscription.

validate(*task*, *method*, ***kwargs*)
Validate vendor-specific actions.

Checks if a valid vendor passthru method was passed and validates the parameters for the vendor passthru method.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **method** method to be validated.
- **kwargs** kwargs containing the vendor passthru methods parameters.

Raises InvalidParameterValue, if any of the parameters have invalid value.

Module contents

ironic.drivers.modules.storage package

Submodules

ironic.drivers.modules.storage.cinder module

class ironic.drivers.modules.storage.cinder.**CinderStorage**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.StorageInterface*

A storage_interface driver supporting Cinder.

attach_volumes(*task*)
Informs the storage subsystem to attach all volumes for the node.

Parameters **task** The task object.

Raises StorageError If an underlying exception or failure is detected.

detach_volumes(*task*, *connector=None*, *aborting_attach=False*)
Informs the storage subsystem to detach all volumes for the node.

This action is retried in case of failure.

Parameters

- **task** The task object.
- **connector** The dictionary representing a nodes connectivity as defined by _generate_connector(). Generated if not passed.
- **aborting_attach** Boolean representing if this detachment was requested to handle aborting a failed attachment

Raises StorageError If an underlying exception or failure is detected.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

should_write_image(*task*)
Determines if deploy should perform the image write-out.

Parameters **task** The task object.

Returns True if the deployment write-out process should be executed.

validate(*task*)
Validate storage_interface configuration for Cinder usage.

In order to provide fail fast functionality prior to nodes being requested to enter the active state, this method performs basic checks of the volume connectors, volume targets, and operator defined capabilities. These checks are to help ensure that we should have a compatible configuration prior to activating the node.

Parameters **task** The task object.

Raises InvalidParameterValue If a misconfiguration or mismatch exists that would prevent storage the cinder storage driver from initializing attachments.

ironic.drivers.modules.storage.external module

class ironic.drivers.modules.storage.external.**ExternalStorage**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.StorageInterface*

Externally driven Storage Interface.

attach_volumes(*task*)
Informs the storage subsystem to attach all volumes for the node.

Parameters **task** A TaskManager instance.

Raises UnsupportedDriverExtension

detach_volumes(*task*)
Informs the storage subsystem to detach all volumes for the node.

Parameters **task** A TaskManager instance.

Raises UnsupportedDriverExtension

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

should_write_image(*task*)
Determines if deploy should perform the image write-out.

This enables the user to define a volume and Ironic understand that the image may already exist and we may be booting to that volume.

Parameters **task** The task object.

Returns True if the deployment write-out process should be executed.

validate(*task*)
Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

ironic.drivers.modules.storage.noop module

class ironic.drivers.modules.storage.noop.**NoopStorage**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.StorageInterface*

No-op Storage Interface.

attach_volumes(*task*)
Informs the storage subsystem to attach all volumes for the node.

Parameters **task** A TaskManager instance.

Raises UnsupportedDriverExtension

detach_volumes(*task*)
Informs the storage subsystem to detach all volumes for the node.

Parameters **task** A TaskManager instance.

Raises UnsupportedDriverExtension

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

should_write_image(*task*)
Determines if deploy should perform the image write-out.

Parameters **task** A TaskManager instance.

Returns Boolean value to indicate if the interface expects the image to be written by Ironic.

Raises UnsupportedDriverExtension

validate(*task*)
Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

Module contents

ironic.drivers.modules.xclarity package

Submodules

ironic.drivers.modules.xclarity.common module

ironic.drivers.modules.xclarity.common.**get_properties**()

ironic.drivers.modules.xclarity.common.**get_server_hardware_id**(*node*)
Validates node configuration and returns xclarity hardware id.

Validates whether node configutation is consistent with XClarity and returns the XClarity Hardware ID for a specific node. :param node: node object to get information from :returns: the XClarity Hardware ID for a specific node :raises: MissingParameterValue if unable to validate XClarity Hardware ID

ironic.drivers.modules.xclarity.common.**get_xclarity_client**(*node*)
Generates an instance of the XClarity client.

Generates an instance of the XClarity client using the imported xclarity_client library.

Parameters **node** an ironic node object.

Returns an instance of the XClarity client

Raises XClarityError if cant get to the XClarity client

ironic.drivers.modules.xclarity.common.**parse_driver_info**(*node*)
Parse a nodes driver_info values.

Parses the driver_info of the node, reads default values and returns a dict containing the combination of both.

Parameters **node** an ironic node object to get informatin from.

Returns a dict containing information parsed from driver_info.

Raises InvalidParameterValue if some required information is missing on the node or inputs is invalid.

ironic.drivers.modules.xclarity.common.**translate_xclarity_power_action**(*power_action*)
Translates ironics power action strings to XClaritys format.

Parameters **power_action** power action string to be translated

Returns the power action translated

ironic.drivers.modules.xclarity.common.**translate_xclarity_power_state**(*power_state*)
Translates XClaritys power state strings to be consistent with Ironic.

Parameters **power_state** power state string to be translated

Returns the translated power state

ironic.drivers.modules.xclarity.management module

class ironic.drivers.modules.xclarity.management.**XClarityManagement**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.ManagementInterface*

get_boot_device(*task*)
Get the current boot device for the tasks node.

Parameters **task** a task from TaskManager.

Returns a dictionary containing: :boot_device: the boot device, one of [PXE, DISK, CDROM, BIOS] :persistent: Whether the boot device will persist or not It returns None if boot device is unknown.

Raises InvalidParameterValue if the boot device is unknown

Raises XClarityError if the communication with XClarity fails

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

get_sensors_data(*task*)
Get sensors data.

Parameters **task** a TaskManager instance.

Raises NotImplementedError

get_supported_boot_devices(*task*)
Gets a list of the supported boot devices.

Parameters **task** a task from TaskManager.

Returns A list with the supported boot devices defined in *ironic.common.boot_devices*.

set_boot_device(*task*, *device*, *persistent=False*)
Sets the boot device for a node.

Parameters

- **task** a task from TaskManager.
- **device** the boot device, one of the supported devices listed in *ironic.common.boot_devices*.
- **persistent** Boolean value. True if the boot device will persist to all future boots, False if not. Default: False.

Raises InvalidParameterValue if an invalid boot device is specified.

Raises XClarityError if the communication with XClarity fails

validate(*task*)
Validate the driver-specific info supplied.

This method validates if the driver_info property of the supplied tasks node contains the required information for this driver to manage the node.

Parameters **task** a task from TaskManager.

ironic.drivers.modules.xclarity.power module

class ironic.drivers.modules.xclarity.power.**XClarityPower**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.PowerInterface*

get_power_state(*task*)
Gets the current power state.

Parameters **task** a TaskManager instance.

Returns one of *ironic.common.states* POWER_OFF, POWER_ON or ERROR.

Raises XClarityError if fails to retrieve power state of XClarity resource

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

reboot(*task*, *timeout=None*)
Soft reboot the node

Parameters

- **task** a TaskManager instance.
- **timeout** timeout (in seconds). Unsupported by this interface.

set_power_state(*task*, *power_state*, *timeout=None*)
Turn the current power state on or off.

Parameters

- **task** a TaskManager instance.
- **power_state** The desired power state POWER_ON, POWER_OFF or REBOOT from *ironic.common.states*.
- **timeout** timeout (in seconds). Unsupported by this interface.

Raises InvalidParameterValue if an invalid power state was specified.

Raises XClarityError if XClarity fails setting the power state.

validate(*task*)

Validate the driver-specific info supplied.

This method validates if the driver_info property of the supplied tasks node contains the required information for this driver to manage the power state of the node.

Parameters **task** a task from TaskManager.

Module contents

Submodules

ironic.drivers.modules.agent module

class ironic.drivers.modules.agent.**AgentDeploy**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.agent.CustomAgentDeploy*

Interface for deploy-related actions.

prepare_instance_boot(*task*)

Prepare instance for booting.

The base version only calls prepare_instance on the boot interface.

validate(*task*)

Validate the driver-specific Node deployment info.

This method validates whether the properties of the supplied node contain the required information for this driver to deploy images to the node.

Parameters **task** a TaskManager instance

Raises MissingParameterValue, if any of the required parameters are missing.

Raises InvalidParameterValue, if any of the parameters have invalid value.

write_image(*task*)

class ironic.drivers.modules.agent.**AgentRAID**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.RAIDInterface*

Implementation of RAIDInterface which uses agent ramdisk.

apply_configuration(*task*, *raid_config*, *delete_existing=True*)

Applies RAID configuration on the given node.

Parameters

- **task** A TaskManager instance.

- **raid_config** The RAID configuration to apply.
- **delete_existing** Setting this to True indicates to delete RAID configuration prior to creating the new configuration.

Raises InvalidParameterValue, if the RAID configuration is invalid.

Returns states.DEPLOYWAIT if RAID configuration is in progress asynchronously or None if it is complete.

create_configuration(*task*, *create_root_volume=True*, *create_nonroot_volumes=True*)
Create a RAID configuration on a bare metal using agent ramdisk.

This method creates a RAID configuration on the given node.

Parameters

- **task** a TaskManager instance.
- **create_root_volume** If True, a root volume is created during RAID configuration. Otherwise, no root volume is created. Default is True.
- **create_nonroot_volumes** If True, non-root volumes are created. If False, no non-root volumes are created. Default is True.

Returns states.CLEANWAIT if operation was successfully invoked.

Raises MissingParameterValue, if node.target_raid_config is missing or was found to be empty after skipping root volume and/or non-root volumes.

delete_configuration(*task*)
Deletes RAID configuration on the given node.

Parameters **task** a TaskManager instance.

Returns states.CLEANWAIT if operation was successfully invoked

get_clean_steps(*task*)
Get the list of clean steps from the agent.

Parameters **task** a TaskManager object containing the node

Raises *NodeCleaningFailure* if the clean steps are not yet available (cached), for example, when a node has just been enrolled and has not been cleaned yet.

Returns A list of clean step dictionaries

get_deploy_steps(*task*)
Get the list of deploy steps from the agent.

Parameters **task** a TaskManager object containing the node

Raises *InstanceDeployFailure* if the deploy steps are not yet available (cached), for example, when a node has just been enrolled and has not been deployed yet.

Returns A list of deploy step dictionaries

get_properties()
Return the properties of the interface.

class ironic.drivers.modules.agent.**AgentRescue**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.RescueInterface*

Implementation of RescueInterface which uses agent ramdisk.

clean_up(*task*)
Clean up after RESCUEWAIT timeout/failure or finishing rescue.

Rescue password should be removed from the node and ramdisk boot environment should be cleaned if Ironic is managing the ramdisk boot.

Parameters **task** a TaskManager instance with the node.

Raises NetworkError if the rescue ports cannot be removed.

get_properties()
Return the properties of the interface.

rescue(*task*)
Boot a rescue ramdisk on the node.

Parameters **task** a TaskManager instance.

Raises NetworkError if the tenant ports cannot be removed.

Raises InvalidParameterValue when the wrong power state is specified or the wrong driver info is specified for power management.

Raises other exceptions by the nodes power driver if something wrong occurred during the power action.

Raises any boot interfaces prepare_ramdisk exceptions.

Returns Returns states.RESCUEWAIT

unrescue(*task*)
Attempt to move a rescued node back to active state.

Parameters **task** a TaskManager instance.

Raises NetworkError if the rescue ports cannot be removed.

Raises InvalidParameterValue when the wrong power state is specified or the wrong driver info is specified for power management.

Raises other exceptions by the nodes power driver if something wrong occurred during the power action.

Raises any boot interfaces prepare_instance exceptions.

Returns Returns states.ACTIVE

validate(*task*)
Validate that the node has required properties for agent rescue.

Parameters **task** a TaskManager instance with the node being checked

Raises InvalidParameterValue if instance_info/rescue_password has empty password or rescuing network UUID config option has an invalid value.

Raises MissingParameterValue if node is missing one or more required parameters

class ironic.drivers.modules.agent.**CustomAgentDeploy**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.agent_base.AgentBaseMixin*, *ironic.drivers.modules.agent_base.AgentDeployMixin*, *ironic.drivers.base.DeployInterface*

A deploy interface that relies on a custom agent to deploy.

Only provides the basic deploy steps to start the ramdisk, tear down the ramdisk and prepare the instance boot.

clean_up(*task*)
Clean up the deployment environment for this node.

If preparation of the deployment environment ahead of time is possible, this method should be implemented by the driver. It should erase anything cached by the *prepare* method.

If implemented, this method must be idempotent. It may be called multiple times for the same node on the same conductor, and it may be called by multiple conductors in parallel. Therefore, it must not require an exclusive lock.

This method is called before *tear_down*.

Parameters **task** a TaskManager instance.

deploy(*task*)
Perform a deployment to a node.

Perform the necessary work to deploy an image onto the specified node. This method will be called after prepare(), which may have already performed any preparatory steps, such as pre-caching some data for the node.

Parameters **task** a TaskManager instance.

Returns status of the deploy. One of ironic.common.states.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

prepare(*task*)
Prepare the deployment environment for this node.

Parameters **task** a TaskManager instance.

Raises NetworkError: if the previous cleaning ports cannot be removed or if new cleaning ports cannot be created.

Raises InvalidParameterValue when the wrong power state is specified or the wrong driver info is specified for power management.

Raises StorageError If the storage driver is unable to attach the configured volumes.

Raises other exceptions by the nodes power driver if something wrong occurred during the power action.

Raises exception.ImageRefValidationFailed if image_source is not Glance href and is not HTTP(S) URL.

Raises exception.InvalidParameterValue if network validation fails.

Raises any boot interfaces prepare_ramdisk exceptions.

prepare_instance_boot(*task*)
Prepare instance for booting.

The base version only calls prepare_instance on the boot interface.

should_manage_boot(*task*)
Whether agent boot is managed by ironic.

validate(*task*)
Validate the driver-specific Node deployment info.

This method validates whether the properties of the supplied node contain the required information for this driver to deploy images to the node.

Parameters **task** a TaskManager instance

Raises MissingParameterValue, if any of the required parameters are missing.

Raises InvalidParameterValue, if any of the parameters have invalid value.

ironic.drivers.modules.agent.**check_image_size**(*task*)
Check if the requested image is larger than the ram size.

Parameters **task** a TaskManager instance containing the node to act on.

Raises InvalidParameterValue if size of the image is greater than the available ram size.

ironic.drivers.modules.agent.**validate_http_provisioning_configuration**(*node*)
Validate configuration options required to perform HTTP provisioning.

Parameters **node** an ironic node object

Raises MissingParameterValue if required option(s) is not set.

ironic.drivers.modules.agent.**validate_image_proxies**(*node*)
Check that the provided proxy parameters are valid.

Parameters **node** an Ironic node.

Raises InvalidParameterValue if any of the provided proxy parameters are incorrect.

ironic.drivers.modules.agent_base module

class ironic.drivers.modules.agent_base.**AgentBaseMixin**
Bases: object

Mixin with base methods not relying on any deploy steps.

Provides full support for in-band and out-of-band cleaning and the machinery to support both deploy and clean in-band steps.

clean_up(*task*)
Clean up the deployment environment for the tasks node.

Unlinks TFTP and instance images and triggers image cache cleanup. Removes the TFTP configuration files for this node.

Parameters **task** a TaskManager instance containing the node to act on.

execute_clean_step(*task*, *step*)
Execute a clean step asynchronously on the agent.

Parameters

- **task** a TaskManager object containing the node
- **step** a clean step dictionary to execute

Raises NodeCleaningFailure if the agent does not return a command status

Returns states.CLEANWAIT to signify the step will be completed async

get_clean_steps(*task*)
Get the list of clean steps from the agent.

Parameters **task** a TaskManager object containing the node

Raises *NodeCleaningFailure* if the clean steps are not yet available (cached), for example, when a node has just been enrolled and has not been cleaned yet.

Returns A list of clean step dictionaries

prepare_cleaning(*task*)
Boot into the agent to prepare for cleaning.

Parameters **task** a TaskManager object containing the node

Raises NodeCleaningFailure, NetworkError if the previous cleaning ports cannot be removed or if new cleaning ports cannot be created.

Raises InvalidParameterValue if cleaning network UUID config option has an invalid value.

Returns states.CLEANWAIT to signify an asynchronous prepare

process_next_step(*task*, *step_type*, ***kwargs*)
Start the next clean/deploy step if the previous one is complete.

In order to avoid errors and make agent upgrades painless, the agent compares the version of all hardware managers at the start of the process (the agents get_clean|deploy_steps() call) and before executing each step. If the version has changed between steps, the agent is unable to tell if an ordering change will cause an issue so it returns VERSION_MISMATCH. For automated cleaning, we restart the entire cleaning cycle. For manual cleaning or deploy, we dont.

Additionally, if a step includes the reboot_requested property set to True, this method will coordinate the reboot once the step is completed.

refresh_steps(*task*, *step_type*)
Refresh the nodes cached clean/deploy steps from the booted agent.

Gets the nodes steps from the booted agent and caches them. The steps are cached to make get_clean_steps() calls synchronous, and should be refreshed as soon as the agent boots to start cleaning/deploy or if cleaning is restarted because of a hardware manager version mismatch.

Parameters

- **task** a TaskManager instance
- **step_type** clean or deploy

Raises NodeCleaningFailure or InstanceDeployFailure if the agent returns invalid results

should_manage_boot(*task*)
Whether agent boot is managed by ironic.

take_over(*task*)
Take over management of this node from a dead conductor.

Parameters **task** a TaskManager instance.

tear_down(*task*)
Tear down a previous deployment on the tasks node.

Power off the node. All actual clean-up is done in the clean_up() method which should be called separately.

Parameters **task** a TaskManager instance containing the node to act on.

Returns deploy state DELETED.

Raises NetworkError if the cleaning ports cannot be removed.

Raises InvalidParameterValue when the wrong state is specified or the wrong driver info is specified.

Raises StorageError when volume detachment fails.

Raises other exceptions by the nodes power driver if something wrong occurred during the power action.

tear_down_cleaning(*task*)
Clean up the PXE and DHCP files after cleaning.

Parameters **task** a TaskManager object containing the node

Raises NodeCleaningFailure, NetworkError if the cleaning ports cannot be removed

class ironic.drivers.modules.agent_base.**AgentDeployMixin**
Bases: *ironic.drivers.modules.agent_base.HeartbeatMixin*, *ironic.drivers.modules.agent_base.AgentOobStepsMixin*

Mixin with deploy methods.

configure_local_boot(*task*, *root_uuid=None*, *efi_system_part_uuid=None*, *prep_boot_part_uuid=None*)
Helper method to configure local boot on the node.

This method triggers bootloader installation on the node. On successful installation of bootloader, this method sets the node to boot from disk.

Parameters

- **task** a TaskManager object containing the node
- **root_uuid** The UUID of the root partition. This is used for identifying the partition which contains the image deployed or None in case of whole disk images which we expect to already have a bootloader installed.
- **efi_system_part_uuid** The UUID of the efi system partition. This is used only in uefi boot mode.
- **prep_boot_part_uuid** The UUID of the PReP Boot partition. This is used only for booting ppc64* hardware.

Raises InstanceDeployFailure if bootloader installation failed or on encountering error while setting the boot device on the node.

execute_deploy_step(*task*, *step*)
Execute a deploy step.

Were trying to find a step among both out-of-band and in-band steps. In case of duplicates, out-of-band steps take priority. This property allows having an out-of-band deploy step that calls into a corresponding in-band step after some preparation (e.g. with additional input).

Parameters

- **task** a TaskManager object containing the node
- **step** a deploy step dictionary to execute

Raises InstanceDeployFailure if the agent does not return a command status

Returns states.DEPLOYWAIT to signify the step will be completed async

get_deploy_steps(*task*)

Get the list of deploy steps from the agent.

Parameters **task** a TaskManager object containing the node

Raises *InstanceDeployFailure* if the deploy steps are not yet available (cached), for example, when a node has just been enrolled and has not been deployed yet.

Returns A list of deploy step dictionaries

prepare_instance_to_boot(*task*, *root_uuid*, *efi_sys_uuid*, *prep_boot_part_uuid=None*)

Prepares instance to boot.

Parameters

- **task** a TaskManager object containing the node
- **root_uuid** the UUID for root partition
- **efi_sys_uuid** the UUID for the efi partition

Raises InvalidState if fails to prepare instance

tear_down_agent(*task*)

A deploy step to tear down the agent.

Parameters **task** a TaskManager object containing the node

class ironic.drivers.modules.agent_base.**AgentOobStepsMixin**

Bases: object

Mixin with out-of-band deploy steps.

boot_instance(*task*)

Deploy step to boot the final instance.

Parameters **task** a TaskManager object containing the node

switch_to_tenant_network(*task*)

Deploy step to switch the node to the tenant network.

Parameters **task** a TaskManager object containing the node

class ironic.drivers.modules.agent_base.**HeartbeatMixin**

Bases: object

Mixin class implementing heartbeat processing.

collect_deploy_logs = True

continue_cleaning(*task*)
Start the next cleaning step if the previous one is complete.

Parameters **task** a TaskManager instance

heartbeat(*task*, *callback_url*, *agent_version*, *agent_verify_ca=None*, *agent_status=None*, *agent_status_message=None*)
Process a heartbeat.

Parameters

- **task** task to work with.
- **callback_url** agent HTTP API URL.
- **agent_version** The version of the agent that is heartbeating
- **agent_verify_ca** TLS certificate for the agent.
- **agent_status** Status of the heartbeating agent
- **agent_status_message** Status message that describes the agent_status

heartbeat_allowed(*node*)

process_next_step(*task*, *step_type*)
Start the next clean/deploy step if the previous one is complete.

Parameters

- **task** a TaskManager instance
- **step_type** clean or deploy

reboot_to_instance(*task*)
Method invoked after the deployment is completed.

Parameters **task** a TaskManager instance

refresh_clean_steps(*task*)
Refresh the nodes cached clean steps

Parameters **task** a TaskManager instance

refresh_steps(*task*, *step_type*)
Refresh the nodes cached clean steps

Parameters

- **task** a TaskManager instance
- **step_type** clean or deploy

ironic.drivers.modules.agent_base.**execute_clean_step**(*task*, *step*)

ironic.drivers.modules.agent_base.**execute_step**(*task*, *step*, *step_type*, *client=None*)
Execute a clean or deploy step asynchronously on the agent.

Parameters

- **task** a TaskManager object containing the node
- **step** a step dictionary to execute
- **step_type** clean or deploy

- **client** agent client (if available)

Raises NodeCleaningFailure (clean step) or InstanceDeployFailure (deploy step) if the agent does not return a command status.

Returns states.CLEANWAIT/DEPLOYWAIT to signify the step will be completed async

ironic.drivers.modules.agent_base.**find_step**(*task*, *step_type*, *interface*, *name*)

Find the given in-band step.

ironic.drivers.modules.agent_base.**get_steps**(*task*, *step_type*, *interface=None*, *override_priorities=None*)

Get the list of cached clean or deploy steps from the agent.

The steps cache is updated at the beginning of cleaning or deploy.

Parameters

- **task** a TaskManager object containing the node
- **step_type** clean or deploy
- **interface** The interface for which clean/deploy steps are to be returned. If this is not provided, it returns the steps for all interfaces.
- **override_priorities** a dictionary with keys being step names and values being new priorities for them. If a step isnt in this dictionary, the steps original priority is used.

Returns A list of clean/deploy step dictionaries

ironic.drivers.modules.agent_base.**log_and_raise_deployment_error**(*task*, *msg*, *collect_logs=True*, *exc=None*)

Helper method to log the error and raise exception.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **msg** the message to set in last_error of the node.
- **collect_logs** Boolean indicating whether to attempt to collect logs from IPA-based ramdisk. Defaults to True. Actual log collection is also affected by CONF.agent.deploy_logs_collect config option.
- **exc** Exception that caused the failure.

ironic.drivers.modules.agent_base.**post_clean_step_hook**(*interface*, *step*)

Decorator method for adding a post clean step hook.

This is a mechanism for adding a post clean step hook for a particular clean step. The hook will get executed after the clean step gets executed successfully. The hook is not invoked on failure of the clean step.

Any method to be made as a hook may be decorated with @post_clean_step_hook mentioning the interface and step after which the hook should be executed. A TaskManager instance and the object for the last completed command (provided by agent) will be passed to the hook method. The return value of this method will be ignored. Any exception raised by this method will be treated as a failure of the clean step and the node will be moved to CLEANFAIL state.

Parameters

- **interface** name of the interface
- **step** The name of the step after which it should be executed.

Returns A method which registers the given method as a post clean step hook.

ironic.drivers.modules.agent_base.**post_deploy_step_hook**(*interface*, *step*)

Decorator method for adding a post deploy step hook.

This is a mechanism for adding a post deploy step hook for a particular deploy step. The hook will get executed after the deploy step gets executed successfully. The hook is not invoked on failure of the deploy step.

Any method to be made as a hook may be decorated with @post_deploy_step_hook mentioning the interface and step after which the hook should be executed. A TaskManager instance and the object for the last completed command (provided by agent) will be passed to the hook method. The return value of this method will be ignored. Any exception raised by this method will be treated as a failure of the deploy step and the node will be moved to DEPLOYFAIL state.

Parameters

- **interface** name of the interface
- **step** The name of the step after which it should be executed.

Returns A method which registers the given method as a post deploy step hook.

ironic.drivers.modules.agent_client module

class ironic.drivers.modules.agent_client.**AgentClient**

Bases: object

Client for interacting with nodes via a REST API.

collect_system_logs(*node*)

Collect and package diagnostic and support data from the ramdisk.

Parameters **node** A Node object.

Raises IronicException when failed to issue the request or there was a malformed response from the agent.

Raises AgentAPIError when agent failed to execute specified command.

Raises AgentInProgress when the command fails to execute as the agent is presently executing the prior command.

Returns A dict containing command response from agent. See *get_commands_status()* for a command result sample.

execute_clean_step(*step*, *node*, *ports*)

Execute specified clean step.

Parameters

- **step** A clean step dictionary to execute.
- **node** A Node object.

- **ports** Ports associated with the node.

Raises IronicException when failed to issue the request or there was a malformed response from the agent.

Raises AgentAPIError when agent failed to execute specified command.

Raises AgentInProgress when the command fails to execute as the agent is presently executing the prior command.

Returns

A dict containing command response from agent. See *get_commands_status()* for a command result sample. The value of key command_result is in the form of:

```
{
  'clean_result': <the result of execution, step specific>,
  'clean_step': <the clean step issued to agent>
}
```

execute_deploy_step(*step*, *node*, *ports*)

Execute specified deploy step.

Parameters

- **step** A deploy step dictionary to execute.
- **node** A Node object.
- **ports** Ports associated with the node.

Raises IronicException when failed to issue the request or there was a malformed response from the agent.

Raises AgentAPIError when agent failed to execute specified command.

Raises AgentInProgress when the command fails to execute as the agent is presently executing the prior command.

Returns

A dict containing command response from agent. See *get_commands_status()* for a command result sample. The value of key command_result is in the form of:

```
{
  'deploy_result': <the result of execution, step specific>
↪,
  'deploy_step': <the deploy step issued to agent>
}
```

finalize_rescue(*node*)

Instruct the ramdisk to finalize entering of rescue mode.

Parameters **node** A Node object.

Raises IronicException if rescue_password is missing, or when failed to issue the request, or there was a malformed response from the agent.

Raises AgentAPIError when agent failed to execute specified command.

Raises AgentInProgress when the command fails to execute as the agent is presently executing the prior command.

Raises InstanceRescueFailure when the agent ramdisk is too old to support transmission of the rescue password.

Returns A dict containing command response from agent. See *get_commands_status()* for a command result sample.

get_clean_steps(*node*, *ports*)

Get clean steps from agent.

Parameters

- **node** A node object.
- **ports** Ports associated with the node.

Raises IronicException when failed to issue the request or there was a malformed response from the agent.

Raises AgentAPIError when agent failed to execute specified command.

Raises AgentInProgress when the command fails to execute as the agent is presently executing the prior command.

Returns

A dict containing command response from agent. See *get_commands_status()* for a command result sample. The value of key command_result is in the form of:

```
{
  'clean_steps': <a list of clean steps>,
  'hardware_manager_version': <manager version>
}
```

get_commands_status(*node*, *retry_connection=True*, *expect_errors=False*)

Get command status from agent.

Parameters

- **node** A Node object.
- **retry_connection** Whether to retry connection problems.
- **expect_errors** If True, do not log connection problems as errors.

Returns

A list of command results, each result is related to a command been issued to agent. A typical result can be:

```
{
  'command_name': <command name related to the result>,
  'command_params': <params related with the command>,
  'command_status': <current command status,
                     e.g. 'RUNNING', 'SUCCEEDED', 'FAILED'>,
```

(continues on next page)

(continued from previous page)

```
  'command_error': <error message if command execution
                   failed>,
  'command_result': <command result if command execution
                    succeeded, the value is command␣
↪specific,
                    e.g.:
                    * a dictionary containing keys clean_
↪result
                      and clean_step for the command
                      clean.execute_clean_step;
                    * a dictionary containing keys deploy_
↪result
                      and deploy_step for the command
                      deploy.execute_deploy_step;
                    * a string representing result message␣
↪for
                      the command standby.cache_image;
                    * None for the command standby.sync.>
}
```

get_deploy_steps(*node*, *ports*)

Get deploy steps from agent.

Parameters

- **node** A node object.
- **ports** Ports associated with the node.

Raises IronicException when failed to issue the request or there was a malformed response from the agent.

Raises AgentAPIError when agent failed to execute specified command.

Returns

A dict containing command response from agent. See *get_commands_status()* for a command result sample. The value of key command_result is in the form of:

```
{
  'deploy_steps': <a list of deploy steps>,
  'hardware_manager_version': <manager version>
}
```

get_last_command_status(*node*, *method*)

Get the last status for the given command.

Parameters

- **node** A Node object.
- **method** Command name.

Returns A dict containing command status from agent or None if the command was not found.

get_partition_uuids(*node*)
Get deploy steps from agent.

Parameters **node** A node object.

Raises IronicException when failed to issue the request or there was a malformed response from the agent.

Raises AgentAPIError when agent failed to execute specified command.

Raises AgentInProgress when the command fails to execute as the agent is presently executing the prior command.

Returns A dict containing command response from agent.

install_bootloader(*node*, *root_uuid*, *target_boot_mode*, *efi_system_part_uuid=None*, *prep_boot_part_uuid=None*, *software_raid=False*)
Install a boot loader on the image.

Parameters

- **node** A node object.
- **root_uuid** The UUID of the root partition.
- **target_boot_mode** The target deployment boot mode.
- **efi_system_part_uuid** The UUID of the efi system partition where the bootloader will be installed to, only used for uefi boot mode.
- **prep_boot_part_uuid** The UUID of the PReP Boot partition where the bootloader will be installed to when local booting a partition image on a ppc64* system.

Raises IronicException when failed to issue the request or there was a malformed response from the agent.

Raises AgentAPIError when agent failed to execute specified command.

Raises AgentInProgress when the command fails to execute as the agent is presently executing the prior command.

Returns A dict containing command response from agent. See *get_commands_status()* for a command result sample.

power_off(*node*)
Soft powers off the bare metal node by shutting down ramdisk OS.

Parameters **node** A Node object.

Raises IronicException when failed to issue the request or there was a malformed response from the agent.

Raises AgentAPIError when agent failed to execute specified command.

Raises AgentInProgress when the command fails to execute as the agent is presently executing the prior command.

Returns A dict containing command response from agent. See *get_commands_status()* for a command result sample.

reboot(*node*)
Soft reboots the bare metal node by shutting down ramdisk OS.

Parameters **node** A Node object.

Raises IronicException when failed to issue the request or there was a malformed response from the agent.

Raises AgentAPIError when agent failed to execute specified command.

Raises AgentInProgress when the command fails to execute as the agent is presently executing the prior command.

Returns A dict containing command response from agent. See *get_commands_status()* for a command result sample.

sync(*node*)
Flush file system buffers forcing changed blocks to disk.

Parameters **node** A Node object.

Raises IronicException when failed to issue the request or there was a malformed response from the agent.

Raises AgentAPIError when agent failed to execute specified command.

Raises AgentInProgress when the command fails to execute as the agent is presently executing the prior command.

Returns A dict containing command response from agent. See *get_commands_status()* for a command result sample.

ironic.drivers.modules.agent_client.**get_client**(*task*)
Get client for this node.

ironic.drivers.modules.agent_client.**get_command_error**(*command*)
Extract an error string from the command result.

Parameters **command** Command information from the agent.

Returns Error string.

ironic.drivers.modules.agent_power module

The agent power interface.

class ironic.drivers.modules.agent_power.**AgentPower**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.PowerInterface*

Power interface using the running agent for power actions.

get_power_state(*task*)
Return the power state of the tasks node.

Essentially, the only known state is POWER ON, everything else is an error (or more precisely `None`).

Parameters **task** A TaskManager instance containing the node to act on.

Returns A power state. One of *ironic.common.states*.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

`get_supported_power_states`(*task*)
Get a list of the supported power states.

Only contains REBOOT.

Parameters `task` A TaskManager instance containing the node to act on.

Returns A list with the supported power states defined in *ironic.common.states*.

`reboot`(*task*, *timeout=None*)
Perform a reboot of the tasks node.

Only soft reboot is implemented.

Parameters

- `task` A TaskManager instance containing the node to act on.
- `timeout` timeout (in seconds) positive integer (> 0) for any power state. `None` indicates to use default timeout.

`set_power_state`(*task*, *power_state*, *timeout=None*)
Set the power state of the tasks node.

Parameters

- `task` A TaskManager instance containing the node to act on.
- `power_state` Power state from *ironic.common.states*. Only REBOOT and SOFT_REBOOT are supported and are synonymous.
- `timeout` timeout (in seconds) positive integer (> 0) for any power state. `None` indicates to use default timeout.

Raises PowerStateFailure on non-supported power state.

`supports_power_sync`(*task*)
Check if power sync is supported for the given node.

Not supported for the agent power since it is not possible to power on/off nodes.

Parameters `task` A TaskManager instance containing the node to act on with a **shared** lock.

Returns boolean, whether power sync is supported.

`validate`(*task*)
Validate the driver-specific Node deployment info.

Parameters `task` A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

ironic.drivers.modules.boot_mode_utils module

ironic.drivers.modules.boot_mode_utils.**configure_secure_boot_if_needed**(*task*)
Configures secure boot if it has been requested for the node.

ironic.drivers.modules.boot_mode_utils.**deconfigure_secure_boot_if_needed**(*task*)
Deconfigures secure boot if it has been requested for the node.

ironic.drivers.modules.boot_mode_utils.**get_boot_mode**(*node*)
Returns the boot mode.

Parameters **node** an ironic node object.

Returns bios or uefi

Raises InvalidParameterValue, if the node boot mode disagrees with the boot mode set to node properties/capabilities

ironic.drivers.modules.boot_mode_utils.**get_boot_mode_for_deploy**(*node*)
Returns the boot mode that would be used for deploy.

This method returns boot mode to be used for deploy. It returns uefi if secure_boot is set to true or returns bios if trusted_boot is set to true in instance_info/capabilities of node. Otherwise it returns value of boot_mode in properties/capabilities of node if set. If that is not set, it returns boot mode in internal_driver_info/deploy_boot_mode for the node. If that is not set, it returns boot mode in instance_info/deploy_boot_mode for the node. It would return None if boot mode is present neither in capabilities of node properties nor in nodes internal_driver_info nor in nodes instance_info (which could also be None).

Parameters **node** an ironic node object.

Returns bios, uefi or None

Raises InvalidParameterValue, if the node boot mode disagrees with the boot mode set to node properties/capabilities

ironic.drivers.modules.boot_mode_utils.**is_secure_boot_requested**(*node*)
Returns True if secure_boot is requested for deploy.

This method checks node property for secure_boot and returns True if it is requested.

Parameters **node** a single Node.

Raises InvalidParameterValue if the capabilities string is not a dictionary or is malformed.

Returns True if secure_boot is requested.

ironic.drivers.modules.boot_mode_utils.**is_trusted_boot_requested**(*node*)
Returns True if trusted_boot is requested for deploy.

This method checks instance property for trusted_boot and returns True if it is requested.

Parameters **node** a single Node.

Raises InvalidParameterValue if the capabilities string is not a dictionary or is malformed.

Returns True if trusted_boot is requested.

ironic.drivers.modules.boot_mode_utils.**sync_boot_mode**(*task*)
Set nodes boot mode from bare metal configuration

Attempt to read currently set boot mode off the bare metal machine. Also read nodes boot mode configuration:

- If BM driver does not implement getting boot mode, assume BM boot mode is not set and apply the logic that follows
- If Ironic node boot mode is not set and BM node boot mode is not set - set Ironic boot mode to *[deploy]/default_boot_mode*
- If Ironic node boot mode is not set and BM node boot mode is set - set BM node boot mode on the Ironic node
- If Ironic node boot mode is set and BM node boot mode is not set - set Ironic boot mode to BM boot mode
- If both Ironic and BM node boot modes are set but they differ - try to set Ironic boot mode to BM boot mode and fail hard if underlying hardware type does not support setting boot mode

In the end, the new boot mode may be set in driver_internal_info/deploy_boot_mode.

Parameters **task** a task object

ironic.drivers.modules.console_utils module

Ironic console utilities.

ironic.drivers.modules.console_utils.**acquire_port**(*host=None*)
Returns a free TCP port on current host.

Find and returns a free TCP port in the range of CONF.console.port_range.

ironic.drivers.modules.console_utils.**get_shellinabox_console_url**(*port*)
Get a url to access the console via shellinaboxd.

Parameters **port** the terminal port for the node.

ironic.drivers.modules.console_utils.**get_socat_console_url**(*port*)
Get a URL to access the console via socat.

Parameters **port** the terminal port (integer) for the node

Returns an access URL to the socat console of the node

ironic.drivers.modules.console_utils.**make_persistent_password_file**(*path*, *password*)
Writes a file containing a password until deleted.

ironic.drivers.modules.console_utils.**release_port**(*port*)
Release specified TCP port.

ironic.drivers.modules.console_utils.**start_shellinabox_console**(*node_uuid*, *port*, *console_cmd*)
Open the serial console for a node.

Parameters

- **node_uuid** the uuid for the node.

- **port** the terminal port for the node.
- **console_cmd** the shell command that gets the console.

Raises ConsoleError if the directory for the PID file cannot be created or an old process cannot be stopped.

Raises ConsoleSubprocessFailed when invoking the subprocess failed.

ironic.drivers.modules.console_utils.**start_socat_console**(*node_uuid*, *port*, *console_cmd*)

Open the serial console for a node.

Parameters

- **node_uuid** the uuid of the node
- **port** the terminal port for the node
- **console_cmd** the shell command that will be executed by socat to establish console to the node

Raises

- *ConsoleError* if the directory for the PID file or the PID file cannot be created
- *ConsoleSubprocessFailed* when invoking the subprocess failed

ironic.drivers.modules.console_utils.**stop_shellinabox_console**(*node_uuid*)

Close the serial console for a node.

Parameters **node_uuid** the UUID of the node

Raises ConsoleError if unable to stop the console process

ironic.drivers.modules.console_utils.**stop_socat_console**(*node_uuid*)

Close the serial console for a node.

Parameters **node_uuid** the UUID of the node

Raises *ConsoleError* if unable to stop the console process

ironic.drivers.modules.deploy_utils module

class ironic.drivers.modules.deploy_utils.**InstanceImageCache**

Bases: *ironic.drivers.modules.image_cache.ImageCache*

ironic.drivers.modules.deploy_utils.**agent_add_clean_params**(*task*)

Add required config parameters to nodes driver_internal_info.

Adds the required conf options to nodes driver_internal_info. It is Required to pass the information to IPA.

Parameters **task** a TaskManager instance.

ironic.drivers.modules.deploy_utils.**build_agent_options**(*node*)

Build the options to be passed to the agent ramdisk.

Parameters **node** an ironic node object

Returns a dictionary containing the parameters to be passed to agent ramdisk.

ironic.drivers.modules.deploy_utils.**build_instance_info_for_deploy**(*task*)
Build instance_info necessary for deploying to a node.

Parameters **task** a TaskManager object containing the node

Returns a dictionary containing the properties to be updated in instance_info

Raises exception.ImageRefValidationFailed if image_source is not Glance href and is not HTTP(S) URL.

ironic.drivers.modules.deploy_utils.**cache_instance_image**(*ctx*, *node*, *force_raw=None*)
Fetch the instances image from Glance

This method pulls the disk image and writes them to the appropriate place on local disk.

Parameters

- **ctx** context
- **node** an ironic node object
- **force_raw** whether convert image to raw format

Returns a tuple containing the uuid of the image and the path in the filesystem where image is cached.

ironic.drivers.modules.deploy_utils.**check_for_missing_params**(*info_dict*, *error_msg*, *param_prefix=''*)
Check for empty params in the provided dictionary.

Parameters

- **info_dict** The dictionary to inspect.
- **error_msg** The error message to prefix before printing the information about missing parameters.
- **param_prefix** Add this prefix to each parameter for error messages

Raises MissingParameterValue, if one or more parameters are empty in the provided dictionary.

ironic.drivers.modules.deploy_utils.**check_interface_capability**(*interface*, *capability*)
Evaluate interface to determine if capability is present.

Parameters

- **interface** The interface object to check.
- **capability** The value representing the capability that the caller wishes to check if present.

Returns True if capability found, otherwise False.

ironic.drivers.modules.deploy_utils.**compute_image_checksum**(*image_path*, *algorithm='md5'*)
Compute checksum by given image path and algorithm.

ironic.drivers.modules.deploy_utils.**destroy_http_instance_images**(*node*)
Delete instance image file and symbolic link refers to it.

ironic.drivers.modules.deploy_utils.**destroy_images**(*node_uuid*)
Delete instances image file.

Parameters **node_uuid** the uuid of the ironic node.

ironic.drivers.modules.deploy_utils.**direct_deploy_should_convert_raw_image**(*node*)
Whether converts image to raw format for specified node.

Parameters **node** ironic node object

Returns Boolean, whether the direct deploy interface should convert image to raw.

ironic.drivers.modules.deploy_utils.**fetch_images**(*ctx*, *cache*, *images_info*, *force_raw=True*)
Check for available disk space and fetch images using ImageCache.

Parameters

- **ctx** context
- **cache** ImageCache instance to use for fetching
- **images_info** list of tuples (image href, destination path)
- **force_raw** boolean value, whether to convert the image to raw format

Raises InstanceDeployFailure if unable to find enough disk space

ironic.drivers.modules.deploy_utils.**get_async_step_return_state**(*node*)
Returns state based on operation (cleaning/deployment) being invoked

Parameters **node** an ironic node object.

Returns states.CLEANWAIT if cleaning operation in progress or states.DEPLOYWAIT if deploy operation in progress.

ironic.drivers.modules.deploy_utils.**get_boot_option**(*node*)
Gets the boot option.

Parameters **node** A single Node.

Raises InvalidParameterValue if the capabilities string is not a dict or is malformed.

Returns A string representing the boot option type. Defaults to configuration setting [deploy]default_boot_mode.

ironic.drivers.modules.deploy_utils.**get_disk_label**(*node*)
Return the disk label requested for deploy, if any.

Parameters **node** a single Node.

Raises InvalidParameterValue if the capabilities string is not a dictionary or is malformed.

Returns the disk label or None if no disk label was specified.

ironic.drivers.modules.deploy_utils.**get_image_download_source**(*node*)
Get the effective value of image_download_source for the node.

ironic.drivers.modules.deploy_utils.**get_image_instance_info**(*node*)
Gets the image information from the node.

Get image information for the given node instance from its instance_info property.

Parameters **node** a single Node.

Returns A dict with required image properties retrieved from nodes instance_info.

Raises MissingParameterValue, if image_source is missing in nodes instance_info. Also raises same exception if kernel/ramdisk is missing in instance_info for non-glance images.

ironic.drivers.modules.deploy_utils.**get_image_properties**(*ctx*, *image_href*)
Get properties of the image.

Parameters

- **ctx** security context
- **image_href** reference to the image

Returns properties as a dictionary

Raises InvalidParameterValue if the image cannot be accessed

ironic.drivers.modules.deploy_utils.**get_ipxe_boot_file**(*node*)
Return the iPXE boot file name requested for deploy.

This method returns iPXE boot file name to be used for deploy. Architecture specific boot file is searched first. BIOS/UEFI boot file is used if no valid architecture specific file found.

If no valid value is found, the default reverts to the `get_pxe_boot_file` method and thus the `[pxe]pxe_bootfile_name` and `[pxe]uefi_ipxe_bootfile_name` settings.

Parameters **node** A single Node.

Returns The iPXE boot file name.

ironic.drivers.modules.deploy_utils.**get_ipxe_config_template**(*node*)
Return the iPXE config template file name requested of deploy.

This method returns the iPXE configuration template file.

Parameters **node** A single Node.

Returns The iPXE config template file name.

ironic.drivers.modules.deploy_utils.**get_ironic_api_url**()
Resolve Ironic API endpoint

either from config of from Keystone catalog.

ironic.drivers.modules.deploy_utils.**get_pxe_boot_file**(*node*)
Return the PXE boot file name requested for deploy.

This method returns PXE boot file name to be used for deploy. Architecture specific boot file is searched first. BIOS/UEFI boot file is used if no valid architecture specific file found.

Parameters **node** A single Node.

Returns The PXE boot file name.

ironic.drivers.modules.deploy_utils.**get_pxe_config_template**(*node*)
Return the PXE config template file name requested for deploy.

This method returns PXE config template file to be used for deploy. First specific pxe template is searched in the node. After that architecture specific template file is searched. BIOS/UEFI template file is used if no valid architecture specific file found.

Parameters **node** A single Node.

Returns The PXE config template file name.

ironic.drivers.modules.deploy_utils.**get_remote_boot_volume**(*task*)
Identify a boot volume from any configured volumes.

Returns None or the volume target representing the volume.

ironic.drivers.modules.deploy_utils.**get_root_device_for_deploy**(*node*)
Get a root device requested for deployment or None.

Raises InvalidParameterValue on invalid hints.

Returns Parsed root device hints or None if no hints were provided.

ironic.drivers.modules.deploy_utils.**get_single_nic_with_vif_port_id**(*task*)
Returns the MAC address of a port which has a VIF port id.

Parameters **task** a TaskManager instance containing the ports to act on.

Returns MAC address of the port connected to deployment network. None if it cannot find any port with vif id.

ironic.drivers.modules.deploy_utils.**is_anaconda_deploy**(*node*)
Determine if Anaconda deploy interface is in use for the deployment.

Parameters **node** A single Node.

Returns A boolean value of True when Anaconda deploy interface is in use otherwise False

ironic.drivers.modules.deploy_utils.**is_iscsi_boot**(*task*)
Return true if booting from an iscsi volume.

ironic.drivers.modules.deploy_utils.**is_ramdisk_deploy**(*node*)

ironic.drivers.modules.deploy_utils.**is_software_raid**(*node*)
Determine if software raid is in use for the deployment.

Parameters **node** A single Node.

Returns A boolean value of True when software raid is in use, otherwise False

ironic.drivers.modules.deploy_utils.**needs_agent_ramdisk**(*node*, *mode='deploy'*)
Checks whether the node requires an agent ramdisk now.

ironic.drivers.modules.deploy_utils.**parse_instance_info**(*node*)
Gets the instance specific Node deployment info.

This method validates whether the instance_info property of the supplied node contains the required information for this driver to deploy images to the node.

Parameters **node** a single Node.

Returns A dict with the instance_info values.

Raises MissingParameterValue, if any of the required parameters are missing.

Raises InvalidParameterValue, if any of the parameters have invalid value.

ironic.drivers.modules.deploy_utils.**populate_storage_driver_internal_info**(*task*)
Set node driver_internal_info for boot from volume parameters.

Parameters **task** a TaskManager object containing the node.

Raises StorageError when a node has an iSCSI or FibreChannel boot volume defined but is not capable to support it.

ironic.drivers.modules.deploy_utils.**prepare_agent_boot**(*task*)

Prepare booting the agent on the node.

Parameters **task** a TaskManager instance.

ironic.drivers.modules.deploy_utils.**prepare_inband_cleaning**(*task*, *manage_boot=True*)

Prepares the node to boot into agent for in-band cleaning.

This method does the following: 1. Prepares the cleaning ports for the bare metal node and updates the clean parameters in nodes driver_internal_info. 2. If manage_boot parameter is set to true, it also calls the prepare_ramdisk method of boot interface to boot the agent ramdisk. 3. Reboots the bare metal node.

Parameters

- **task** a TaskManager object containing the node
- **manage_boot** If this is set to True, this method calls the prepare_ramdisk method of boot interface to boot the agent ramdisk. If False, it skips preparing the boot agent ramdisk using boot interface, and assumes that the environment is setup to automatically boot agent ramdisk every time bare metal node is rebooted.

Returns states.CLEANWAIT to signify an asynchronous prepare.

Raises NetworkError, NodeCleaningFailure if the previous cleaning ports cannot be removed or if new cleaning ports cannot be created.

Raises InvalidParameterValue if cleaning network UUID config option has an invalid value.

ironic.drivers.modules.deploy_utils.**reboot_to_finish_step**(*task*)

Reboot the node into IPA to finish a deploy/clean step.

Parameters **task** a TaskManager instance.

Returns states.CLEANWAIT if cleaning operation in progress or states.DEPLOYWAIT if deploy operation in progress.

ironic.drivers.modules.deploy_utils.**remove_http_instance_symlink**(*node_uuid*)

ironic.drivers.modules.deploy_utils.**rescue_or_deploy_mode**(*node*)

ironic.drivers.modules.deploy_utils.**set_async_step_flags**(*node*, *reboot=None*, *skip_current_step=None*, *polling=None*)

Sets appropriate reboot flags in driver_internal_info based on operation

Parameters

- **node** an ironic node object.
- **reboot** Boolean value to set for nodes driver_internal_info flag cleaning_reboot or deployment_reboot based on cleaning or deployment opera-

tion in progress. If it is None, corresponding reboot flag is not set in nodes driver_internal_info.

- **skip_current_step** Boolean value to set for nodes driver_internal_info flag skip_current_clean_step or skip_current_deploy_step based on cleaning or deployment operation in progress. If it is None, corresponding skip step flag is not set in nodes driver_internal_info.
- **polling** Boolean value to set for nodes driver_internal_info flag deployment_polling or cleaning_polling. If it is None, the corresponding polling flag is not set in the nodes driver_internal_info.

ironic.drivers.modules.deploy_utils.**set_failed_state**(*task*, *msg*, *collect_logs=True*)

Sets the deploy status as failed with relevant messages.

This method sets the deployment as fail with the given message. It sets nodes provision_state to DEPLOYFAIL and updates last_error with the given error message. It also powers off the baremetal node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **msg** the message to set in logs and last_error of the node.
- **collect_logs** Boolean indicating whether to attempt to collect logs from IPA-based ramdisk. Defaults to True. Actual log collection is also affected by CONF.agent.deploy_logs_collect config option.

ironic.drivers.modules.deploy_utils.**switch_pxe_config**(*path*, *root_uuid_or_disk_id*, *boot_mode*, *is_whole_disk_image*, *trusted_boot=False*, *iscsi_boot=False*, *ramdisk_boot=False*, *ipxe_enabled=False*, *anaconda_boot=False*)

Switch a pxe config from deployment mode to service mode.

Parameters

- **path** path to the pxe config file in tftpboot.
- **root_uuid_or_disk_id** root uuid in case of partition image or disk_id in case of whole disk image.
- **boot_mode** if boot mode is uefi or bios.
- **is_whole_disk_image** if the image is a whole disk image or not.
- **trusted_boot** if boot with trusted_boot or not. The usage of is_whole_disk_image and trusted_boot are mutually exclusive. You can have one or neither, but not both.
- **iscsi_boot** if boot is from an iSCSI volume or not.
- **ramdisk_boot** if the boot is to be to a ramdisk configuration.
- **ipxe_enabled** A default False boolean value to tell the method if the caller is using iPXE.

- **anaconda_boot** if the boot is to be to an anaconda configuration.

ironic.drivers.modules.deploy_utils.**tear_down_inband_cleaning**(*task*, *manage_boot=True*)

Tears down the environment setup for in-band cleaning.

This method does the following: 1. Powers off the bare metal node (unless the node is fast tracked or there was a cleaning failure). 2. If manage_boot parameter is set to true, it also calls the clean_up_ramdisk method of boot interface to clean up the environment that was set for booting agent ramdisk. 3. Deletes the cleaning ports which were setup as part of cleaning.

Parameters

- **task** a TaskManager object containing the node
- **manage_boot** If this is set to True, this method calls the clean_up_ramdisk method of boot interface to boot the agent ramdisk. If False, it skips this step.

Raises NetworkError, NodeCleaningFailure if the cleaning ports cannot be removed.

ironic.drivers.modules.deploy_utils.**tear_down_storage_configuration**(*task*)

Clean up storage configuration.

Remove entries from driver_internal_info for storage and deletes the volume targets from the database. This is done to ensure a clean state for the next boot of the machine.

ironic.drivers.modules.deploy_utils.**try_set_boot_device**(*task*, *device*, *persistent=True*)

Tries to set the boot device on the node.

This method tries to set the boot device on the node to the given boot device. Under uefi boot mode, setting of boot device may differ between different machines. IPMI does not work for setting boot devices in uefi mode for certain machines. This method ignores the expected IPMI failure for uefi boot mode and just logs a message. In error cases, it is expected the operator has to manually set the node to boot from the correct device.

Parameters

- **task** a TaskManager object containing the node
- **device** the boot device
- **persistent** Whether to set the boot device persistently

Raises Any exception from set_boot_device except IPMIFailure (setting of boot device using ipmi is expected to fail).

ironic.drivers.modules.deploy_utils.**validate_capabilities**(*node*)

Validates that specified supported capabilities have valid value

This method checks if the any of the supported capability is present in Node capabilities. For all supported capabilities specified for a Node, it validates that it has a valid value. The node can have capability as part of the properties or instance_info or both. Note that the actual value of a capability does not need to be the same in the nodes properties and instance_info.

Parameters **node** an ironic node object.

Raises InvalidParameterValue, if the capability is not set to a valid value.

ironic.drivers.modules.deploy_utils.**validate_image_properties**(*task*, *deploy_info*)

Validate the image.

For Glance images it checks that the image exists in Glance and its properties or deployment info contain the properties passed. If its not a Glance image, it checks that deployment info contains needed properties.

Parameters

- **task** TaskManager instance with a valid node
- **deploy_info** the deploy_info to be validated

Raises InvalidParameterValue if: * connection to glance failed; * authorization for accessing image failed; * HEAD request to image URL failed or returned response code != 200; * HEAD request response does not contain Content-Length header; * the protocol specified in image URL is not supported.

Raises MissingParameterValue if the image doesnt contain the mentioned properties.

ironic.drivers.modules.fake module

Fake driver interfaces used in testing.

This is also an example of some kinds of things which can be done within drivers. For instance, the MultipleVendorInterface class demonstrates how to load more than one interface and wrap them in some logic to route incoming vendor_passthru requests appropriately. This can be useful eg. when mixing functionality between a power interface and a deploy interface, when both rely on separate vendor_passthru methods.

class ironic.drivers.modules.fake.**FakeBIOS**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.BIOSInterface*

Fake implementation of simple BIOSInterface.

apply_configuration(*task*, *settings*)

Validate & apply BIOS settings on the given node.

This method takes the BIOS settings from the settings param and applies BIOS settings on the given node. It may also validate the given bios settings before applying any settings and manage failures when setting an invalid BIOS config. In the case of needing password to update the BIOS config, it will be taken from the driver_info properties. After the BIOS configuration is done, cache_bios_settings will be called to update the nodes BIOS setting table with the BIOS configuration applied on the node.

Parameters

- **task** a TaskManager instance.
- **settings** Dictonary containing the BIOS configuration.

Raises UnsupportedDriverExtension, if the nodes driver doesnt support BIOS configuration.

Raises InvalidParameterValue, if validation of settings fails.

Raises MissingParameterValue, if some required parameters are missing.

Returns states.CLEANWAIT if BIOS configuration is in progress asynchronously or None if it is complete.

cache_bios_settings(*task*)
Store or update BIOS properties on the given node.

This method stores BIOS properties to the bios_settings table during cleaning operation and updates bios_settings table when apply_configuration() and factory_reset() are called to set new BIOS configurations. It will also update the timestamp of each bios setting.

Parameters **task** a TaskManager instance.

Raises UnsupportedDriverExtension, if the nodes driver doesnt support getting BIOS properties from bare metal.

Returns None.

factory_reset(*task*)
Reset BIOS configuration to factory default on the given node.

This method resets BIOS configuration to factory default on the given node. After the BIOS reset action is done, cache_bios_settings will be called to update the nodes BIOS settings table with default bios settings.

Parameters **task** a TaskManager instance.

Raises UnsupportedDriverExtension, if the nodes driver doesnt support BIOS reset.

Returns states.CLEANWAIT if BIOS configuration is in progress asynchronously or None if it is complete.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

validate(*task*)
Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

class ironic.drivers.modules.fake.**FakeBoot**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.BootInterface*

Example implementation of a simple boot interface.

capabilities = ['ipxe_boot', 'pxe_boot']

clean_up_instance(*task*)
Cleans up the boot of instance.

This method cleans up the environment that was setup for booting the instance.

Parameters **task** A task from TaskManager.

Returns None

clean_up_ramdisk(*task*, *mode='deploy'*)
Cleans up the boot of ironic ramdisk.

This method cleans up the environment that was setup for booting the deploy or rescue ramdisk.

Parameters **task** A task from TaskManager.

Returns None

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

prepare_instance(*task*)
Prepares the boot of instance.

This method prepares the boot of the instance after reading relevant information from the nodes database.

Parameters **task** A task from TaskManager.

Returns None

prepare_ramdisk(*task*, *ramdisk_params*, *mode='deploy'*)
Prepares the boot of Ironic ramdisk.

This method prepares the boot of the deploy or rescue ramdisk after reading relevant information from the nodes database.

Parameters

- **task** A task from TaskManager.
- **ramdisk_params** The options to be passed to the ironic ramdisk. Different implementations might want to boot the ramdisk in different ways by passing parameters to them. For example,

  When Agent ramdisk is booted to deploy a node, it takes the parameters ipa-api-url, etc.

  Other implementations can make use of ramdisk_params to pass such information. Different implementations of boot interface will have different ways of passing parameters to the ramdisk.

Returns None

validate(*task*)
Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

class ironic.drivers.modules.fake.**FakeConsole**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.ConsoleInterface*

Example implementation of a simple console interface.

get_console(*task*)

Get connection information about the console.

This method should return the necessary information for the client to access the console.

Parameters **task** A TaskManager instance containing the node to act on.

Returns the console connection information.

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

start_console(*task*)

Start a remote console for the tasks node.

This method should not raise an exception if console already started.

Parameters **task** A TaskManager instance containing the node to act on.

stop_console(*task*)

Stop the remote console session for the tasks node.

Parameters **task** A TaskManager instance containing the node to act on.

validate(*task*)

Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

class ironic.drivers.modules.fake.**FakeDeploy**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.DeployInterface*

Class for a fake deployment driver.

Example implementation of a deploy interface that uses a separate power interface.

clean_up(*task*)

Clean up the deployment environment for the tasks node.

If preparation of the deployment environment ahead of time is possible, this method should be implemented by the driver. It should erase anything cached by the *prepare* method.

If implemented, this method must be idempotent. It may be called multiple times for the same node on the same conductor, and it may be called by multiple conductors in parallel. Therefore, it must not require an exclusive lock.

This method is called before *tear_down*.

Parameters `task` A TaskManager instance containing the node to act on.

`deploy`(*task*)
Perform a deployment to the tasks node.

Perform the necessary work to deploy an image onto the specified node. This method will be called after prepare(), which may have already performed any preparatory steps, such as pre-caching some data for the node.

Parameters `task` A TaskManager instance containing the node to act on.

Returns status of the deploy. One of ironic.common.states.

`get_properties`()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

`prepare`(*task*)
Prepare the deployment environment for the tasks node.

If preparation of the deployment environment ahead of time is possible, this method should be implemented by the driver.

If implemented, this method must be idempotent. It may be called multiple times for the same node on the same conductor.

This method is called before *deploy*.

Parameters `task` A TaskManager instance containing the node to act on.

`take_over`(*task*)
Take over management of this tasks node from a dead conductor.

If conductors hosts maintain a static relationship to nodes, this method should be implemented by the driver to allow conductors to perform the necessary work during the remapping of nodes to conductors when a conductor joins or leaves the cluster.

For example, the PXE driver has an external dependency: Neutron must forward DHCP BOOT requests to a conductor which has prepared the tftpboot environment for the given node. When a conductor goes offline, another conductor must change this setting in Neutron as part of remapping that nodes control to itself. This is performed within the *takeover* method.

Parameters `task` A TaskManager instance containing the node to act on.

`tear_down`(*task*)
Tear down a previous deployment on the tasks node.

Given a node that has been previously deployed to, do all cleanup and tear down necessary to un-deploy that node.

Parameters `task` A TaskManager instance containing the node to act on.

Returns status of the deploy. One of ironic.common.states.

`validate`(*task*)
Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

class ironic.drivers.modules.fake.**FakeInspect**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.InspectInterface*

Example implementation of a simple inspect interface.

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

inspect_hardware(*task*)

Inspect hardware.

Inspect hardware to obtain the essential & additional hardware properties.

Parameters **task** A task from TaskManager.

Raises HardwareInspectionFailure, if unable to get essential hardware properties.

Returns Resulting state of the inspection i.e. states.MANAGEABLE or None.

validate(*task*)

Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

class ironic.drivers.modules.fake.**FakeManagement**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.ManagementInterface*

Example implementation of a simple management interface.

get_boot_device(*task*)

Get the current boot device for a node.

Provides the current boot device of the node. Be aware that not all drivers support this.

Parameters **task** A task from TaskManager.

Raises MissingParameterValue if a required parameter is missing

Returns

A dictionary containing:

boot_device Ahe boot device, one of *ironic.common.boot_devices* or None if it is unknown.

persistent Whether the boot device will persist to all future boots or not, None if it is unknown.

get_indicator_state(*task*, *component*, *indicator*)

Get current state of the indicator of the hardware component.

Parameters

- **task** A task from TaskManager.
- **component** The hardware component, one of *ironic.common.components*.
- **indicator** Indicator ID (as reported by *get_supported_indicators*).

Raises InvalidParameterValue if an invalid component or indicator is specified.

Raises MissingParameterValue if a required parameter is missing

Returns Current state of the indicator, one of *ironic.common.indicator_states*.

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

get_sensors_data(*task*)

Get sensors data method.

Parameters **task** A TaskManager instance.

Raises FailedToGetSensorData when getting the sensor data fails.

Raises FailedToParseSensorData when parsing sensor data fails.

Returns

Returns a consistent format dict of sensor data grouped by sensor type, which can be processed by Ceilometer. eg,

```
{
  'Sensor Type 1': {
    'Sensor ID 1': {
      'Sensor Reading': 'current value',
      'key1': 'value1',
      'key2': 'value2'
    },
    'Sensor ID 2': {
      'Sensor Reading': 'current value',
      'key1': 'value1',
      'key2': 'value2'
    }
  },
  'Sensor Type 2': {
    'Sensor ID 3': {
```

(continues on next page)

(continued from previous page)

```
      'Sensor Reading': 'current value',
      'key1': 'value1',
      'key2': 'value2'
    },
    'Sensor ID 4': {
      'Sensor Reading': 'current value',
      'key1': 'value1',
      'key2': 'value2'
    }
  }
}
```

get_supported_boot_devices(*task*)

Get a list of the supported boot devices.

Parameters **task** A task from TaskManager.

Returns A list with the supported boot devices defined in *ironic.common.boot_devices*.

get_supported_indicators(*task*, *component=None*)

Get a map of the supported indicators (e.g. LEDs).

Parameters

- **task** A task from TaskManager.
- **component** If not *None*, return indicator information for just this component, otherwise return indicators for all existing components.

Returns

A dictionary of hardware components (*ironic.common.components*) as keys with values being dictionaries having indicator IDs as keys and indicator properties as values.

```
{
    'chassis': {
        'enclosure-0': {
            "readonly": true,
            "states": [
                "off",
                "on"
            ]
        }
    },
    'system':
        'blade-A': {
            "readonly": true,
            "states": [
                "pff",
                "on"
            ]
```

(continues on next page)

(continued from previous page)

```
        }
    },
    'drive':
        'ssd0': {
            "readonly": true,
            "states": [
                "off",
                "on"
            ]
        }
    }
}
```

set_boot_device(*task*, *device*, *persistent=False*)

Set the boot device for a node.

Set the boot device to use on next reboot of the node.

Parameters

- **task** A task from TaskManager.
- **device** The boot device, one of *ironic.common.boot_devices*.
- **persistent** Boolean value. True if the boot device will persist to all future boots, False if not. Default: False.

Raises InvalidParameterValue if an invalid boot device is specified.

Raises MissingParameterValue if a required parameter is missing

validate(*task*)

Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

class ironic.drivers.modules.fake.**FakePower**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.PowerInterface*

Example implementation of a simple power interface.

get_power_state(*task*)

Return the power state of the tasks node.

Parameters **task** A TaskManager instance containing the node to act on.

Raises MissingParameterValue if a required parameter is missing.

Returns A power state. One of *ironic.common.states*.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

get_supported_power_states(*task*)
Get a list of the supported power states.

Parameters **task** A TaskManager instance containing the node to act on.

Returns A list with the supported power states defined in *ironic.common.states*.

reboot(*task*, *timeout=None*)
Perform a hard reboot of the tasks node.

Drivers are expected to properly handle case when node is powered off by powering it on.

Parameters

- **task** A TaskManager instance containing the node to act on.
- **timeout** timeout (in seconds) positive integer (> 0) for any power state. None indicates to use default timeout.

Raises MissingParameterValue if a required parameter is missing.

set_power_state(*task*, *power_state*, *timeout=None*)
Set the power state of the tasks node.

Parameters

- **task** A TaskManager instance containing the node to act on.
- **power_state** Any power state from *ironic.common.states*.
- **timeout** timeout (in seconds) positive integer (> 0) for any power state. None indicates to use default timeout.

Raises MissingParameterValue if a required parameter is missing.

validate(*task*)
Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

class ironic.drivers.modules.fake.**FakeRAID**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.RAIDInterface*

Example implementation of simple RAIDInterface.

create_configuration(*task*, *create_root_volume=True*, *create_nonroot_volumes=True*)
Creates RAID configuration on the given node.

This method creates a RAID configuration on the given node. It assumes that the target RAID configuration is already available in node.target_raid_config. Implementations of this interface are supposed to read the RAID configuration from node.target_raid_config. After the RAID configuration is done (either in this method OR in a call-back method), ironic.common.raid.update_raid_info() may be called to sync the nodes RAID-related information with the RAID configuration applied on the node.

Parameters

- **task** A TaskManager instance.
- **create_root_volume** Setting this to False indicates not to create root volume that is specified in the nodes target_raid_config. Default value is True.
- **create_nonroot_volumes** Setting this to False indicates not to create non-root volumes (all except the root volume) in the nodes target_raid_config. Default value is True.
- **delete_existing** Setting this to True indicates to delete RAID configuration prior to creating the new configuration.

Returns states.CLEANWAIT (cleaning) or states.DEPLOYWAIT (deployment) if RAID configuration is in progress asynchronously, or None if it is complete.

delete_configuration(*task*)

Deletes RAID configuration on the given node.

This method deletes the RAID configuration on the give node. After RAID configuration is deleted, node.raid_config should be cleared by the implementation.

Parameters **task** A TaskManager instance.

Returns states.CLEANWAIT (cleaning) or states.DEPLOYWAIT (deployment) if deletion is in progress asynchronously, or None if it is complete.

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

class ironic.drivers.modules.fake.**FakeRescue**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.RescueInterface*

Example implementation of a simple rescue interface.

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

rescue(*task*)

Boot the tasks node into a rescue environment.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InstanceRescueFailure if node validation or rescue operation fails.

Returns states.RESCUEWAIT if rescue is in progress asynchronously or states.RESCUE if it is complete.

unrescue(*task*)

Tear down the rescue environment, and return to normal.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InstanceUnrescueFailure if node validation or unrescue operation fails.

Returns states.ACTIVE if it is successful.

validate(*task*)

Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

class ironic.drivers.modules.fake.**FakeStorage**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.StorageInterface*

Example implementation of simple storage Interface.

attach_volumes(*task*)

Informs the storage subsystem to attach all volumes for the node.

Parameters **task** A TaskManager instance.

Raises UnsupportedDriverExtension

detach_volumes(*task*)

Informs the storage subsystem to detach all volumes for the node.

Parameters **task** A TaskManager instance.

Raises UnsupportedDriverExtension

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

should_write_image(*task*)

Determines if deploy should perform the image write-out.

Parameters **task** A TaskManager instance.

Returns Boolean value to indicate if the interface expects the image to be written by Ironic.

Raises UnsupportedDriverExtension

validate(*task*)

Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

class ironic.drivers.modules.fake.**FakeVendorA**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.VendorInterface*

Example implementation of a vendor passthru interface.

first_method(*task*, *http_method*, *bar*)

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

validate(*task*, *method*, ***kwargs*)

Validate vendor-specific actions.

If invalid, raises an exception; otherwise returns None.

Parameters

- **task** A task from TaskManager.
- **method** Method to be validated
- **kwargs** Info for action.

Raises UnsupportedDriverExtension if method can not be mapped to the supported interfaces.

Raises InvalidParameterValue if kwargs does not contain method.

Raises MissingParameterValue

class ironic.drivers.modules.fake.**FakeVendorB**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.VendorInterface*

Example implementation of a secondary vendor passthru.

fourth_method_shared_lock(*task*, *http_method*, *bar*)

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

second_method(*task*, *http_method*, *bar*)

third_method_sync(*task*, *http_method*, *bar*)

validate(*task*, *method*, ***kwargs*)

Validate vendor-specific actions.

If invalid, raises an exception; otherwise returns None.

Parameters

- **task** A task from TaskManager.
- **method** Method to be validated
- **kwargs** Info for action.

Raises UnsupportedDriverExtension if method can not be mapped to the supported interfaces.

Raises InvalidParameterValue if kwargs does not contain method.

Raises MissingParameterValue

ironic.drivers.modules.image_cache module

Utility for caching master images.

class ironic.drivers.modules.image_cache.**ImageCache**(*master_dir*, *cache_size*, *cache_ttl*)
Bases: object

Class handling access to cache for master images.

clean_up(*amount=None*)
Clean up directory with images, keeping cache of the latest images.

Files with link count >1 are never deleted. Protected by global lock, so that no one messes with master images after we get listing and before we actually delete files.

Parameters **amount** if present, amount of space to reclaim in bytes, cleaning will stop, if this goal was reached, even if it is possible to clean up more files

fetch_image(*href*, *dest_path*, *ctx=None*, *force_raw=True*)
Fetch image by given href to the destination path.

Does nothing if destination path exists and is up to date with cache and href contents. Only creates a hard link (dest_path) to cached image if requested image is already in cache and up to date with href contents. Otherwise downloads an image, stores it in cache and creates a hard link (dest_path) to it.

Parameters

- **href** image UUID or href to fetch
- **dest_path** destination file path
- **ctx** context
- **force_raw** boolean value, whether to convert the image to raw format

ironic.drivers.modules.image_cache.**clean_up_all**()
Clean up all entries from all caches.

ironic.drivers.modules.image_cache.**clean_up_caches**(*ctx*, *directory*, *images_info*)
Explicitly cleanup caches based on their priority (if required).

This cleans up the caches to free up the amount of space required for the images in images_info. The caches are cleaned up one after the other in the order of their priority. If we still cannot free up enough space after trying all the caches, this method throws exception.

Parameters

- **ctx** context
- **directory** the directory (of the cache) to be freed up.
- **images_info** a list of tuples of the form (image_uuid,path) for which space is to be created in cache.

Raises InsufficientDiskSpace exception, if we cannot free up enough space after trying all the caches.

ironic.drivers.modules.image_cache.**cleanup**(*priority*)
Decorator method for adding cleanup priority to a class.

ironic.drivers.modules.image_utils module

class ironic.drivers.modules.image_utils.**ISOImageCache**
Bases: *ironic.drivers.modules.image_cache.ImageCache*

class ironic.drivers.modules.image_utils.**ImageHandler**(*driver*)
Bases: object

publish_image(*image_file*, *object_name*)
Make image file downloadable.

Depending on ironic settings, pushes given file into Swift or copies it over to local HTTP servers document root and returns publicly accessible URL leading to the given file.

Parameters

- **image_file** path to file to publish
- **object_name** name of the published file

Returns a URL to download published file

unpublish_image(*object_name*)
Withdraw the image previously made downloadable.

Depending on ironic settings, removes previously published file from where it has been published - Swift or local HTTP servers document root.

Parameters **object_name** name of the published file (optional)

classmethod **unpublish_image_for_node**(*node*, *prefix=''*, *suffix=''*)
Withdraw the image previously made downloadable.

Depending on ironic settings, removes previously published file from where it has been published - Swift or local HTTP servers document root.

Parameters

- **node** the node for which image was published.
- **prefix** object name prefix.
- **suffix** object name suffix.

update_driver_config(*driver*)

ironic.drivers.modules.image_utils.**cleanup_disk_image**(*task*, *prefix=None*)
Deletes the image if it was created for the node.

Parameters

- **task** an ironic node object.
- **prefix** Prefix to use for the object name.

ironic.drivers.modules.image_utils.**cleanup_floppy_image**(*task*)
Deletes the floppy image if it was created for the node.

Parameters **task** an ironic node object.

ironic.drivers.modules.image_utils.**cleanup_iso_image**(*task*)
Deletes the ISO if it was created for the instance.

Parameters **task** A task from TaskManager.

ironic.drivers.modules.image_utils.**override_api_url**(*params*)

ironic.drivers.modules.image_utils.**prepare_boot_iso**(*task*, *d_info*, *root_uuid=None*)
Prepare boot ISO image

Build bootable ISO out of *[instance_info]/kernel*, *[instance_info]/ramdisk* and *[driver_info]/bootloader* if present. Otherwise, read *kernel_id* and *ramdisk_id* from *[instance_info]/image_source* Glance image metadata.

Push produced ISO image up to Glance and return temporary Swift URL to the image.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **d_info** Deployment information of the node
- **root_uuid** Root UUID

Returns bootable ISO HTTP URL.

Raises MissingParameterValue, if any of the required parameters are missing.

Raises InvalidParameterValue, if any of the parameters have invalid value.

Raises ImageCreationFailed, if creating ISO image failed.

ironic.drivers.modules.image_utils.**prepare_configdrive_image**(*task*, *content*)
Prepare an image with configdrive.

Decodes base64 contents and writes it into a disk image that can be attached e.g. to a virtual USB device. Images stored in Swift are downloaded first.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **content** Config drive as a base64-encoded string.

Raises ImageCreationFailed, if it failed while creating the image.

Raises SwiftOperationError, if any operation with Swift fails.

Returns image URL for the image.

ironic.drivers.modules.image_utils.**prepare_deploy_iso**(*task*, *params*, *mode*, *d_info*)
Prepare deploy or rescue ISO image

Build bootable ISO out of *[driver_info]/deploy_kernel*/*[driver_info]/deploy_ramdisk* or *[driver_info]/rescue_kernel*/*[driver_info]/rescue_ramdisk* and *[driver_info]/bootloader*, then push built image up to Glance and return temporary Swift URL to the image.

If network interface supplies network configuration (*network_data*), a *network_data.json* will be written into an appropriate location on the final ISO.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **params** a dictionary containing parameter name->value mapping to be passed to kernel command line.
- **mode** either deploy or rescue.
- **d_info** Deployment information of the node

Returns bootable ISO HTTP URL.

Raises MissingParameterValue, if any of the required parameters are missing.

Raises InvalidParameterValue, if any of the parameters have invalid value.

Raises ImageCreationFailed, if creating ISO image failed.

ironic.drivers.modules.image_utils.**prepare_disk_image**(*task*, *content*, *prefix=None*)

Prepare an image with the given content.

If content is already an HTTP URL, return it unchanged.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **content** Content as a string with a file name or bytes with contents.
- **prefix** Prefix to use for the object name.

Raises ImageCreationFailed, if it failed while creating the image.

Raises SwiftOperationError, if any operation with Swift fails.

Returns image URL for the image.

ironic.drivers.modules.image_utils.**prepare_floppy_image**(*task*, *params=None*)

Prepares the floppy image for passing the parameters.

This method prepares a temporary VFAT filesystem image and adds a file into the image which contains parameters to be passed to the ramdisk. Then this method uploads built image to Swift [redfish]swift_container, setting it to auto expire after [redfish]swift_object_expiry_timeout seconds. Finally, a temporary Swift URL is returned addressing Swift object just created.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **params** a dictionary containing parameter name->value mapping to be passed to deploy or rescue image via floppy image.

Raises ImageCreationFailed, if it failed while creating the floppy image.

Raises SwiftOperationError, if any operation with Swift fails.

Returns image URL for the floppy image.

ironic.drivers.modules.inspect_utils module

ironic.drivers.modules.inspect_utils.**create_ports_if_not_exist**(*task*, *macs*)
Create ironic ports from MAC addresses data dict.

Creates ironic ports from MAC addresses data returned with inspection or as requested by operator. Helper argument to detect the MAC address `get_mac_address` defaults to value part of MAC address dict key-value pair.

Parameters

- **task** A TaskManager instance.
- **macs** A sequence of MAC addresses.

ironic.drivers.modules.inspector module

Modules required to work with ironic_inspector: https://pypi.org/project/ironic-inspector

class ironic.drivers.modules.inspector.**Inspector**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.InspectInterface*

In-band inspection via ironic-inspector project.

abort(*task*)
Abort hardware inspection.

Parameters **task** a task from TaskManager.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

inspect_hardware(*task*)
Inspect hardware to obtain the hardware properties.

This particular implementation only starts inspection using ironic-inspector. Results will be checked in a periodic task.

Parameters **task** a task from TaskManager.

Returns states.INSPECTWAIT

Raises HardwareInspectionFailure on failure

validate(*task*)
Validate the driver-specific inspection information.

If invalid, raises an exception; otherwise returns None.

Parameters **task** a task from TaskManager.

Raises UnsupportedDriverExtension

ironic.drivers.modules.ipmitool module

IPMI power manager driver.

Uses the ipmitool command (http://ipmitool.sourceforge.net/) to remotely manage hardware. This includes setting the boot device, getting a serial-over-LAN console, and controlling the power state of the machine.

NOTE THAT CERTAIN DISTROS MAY INSTALL openipmi BY DEFAULT, INSTEAD OF ipmitool, WHICH PROVIDES DIFFERENT COMMAND-LINE OPTIONS AND *IS NOT SUPPORTED* BY THIS DRIVER.

class ironic.drivers.modules.ipmitool.**IPMIConsole**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.ConsoleInterface*

A base ConsoleInterface that uses ipmitool.

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

validate(*task*)

Validate the Node console info.

Parameters **task** a task from TaskManager.

Raises InvalidParameterValue

Raises MissingParameterValue when a required parameter is missing

class ironic.drivers.modules.ipmitool.**IPMIManagement**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.ManagementInterface*

detect_vendor(*task*)

Detects and returns the hardware vendor.

Parameters **task** A task from TaskManager.

Raises InvalidParameterValue if an invalid component, indicator or state is specified.

Raises MissingParameterValue if a required parameter is missing

Returns String representing the BMC reported Vendor or Manufacturer, otherwise returns None.

get_boot_device(*task*)

Get the current boot device for the tasks node.

Returns the current boot device of the node.

Parameters **task** a task from TaskManager.

Raises InvalidParameterValue if required IPMI parameters are missing.

Raises IPMIFailure on an error from ipmitool.

Raises MissingParameterValue if a required parameter is missing.

Returns

a dictionary containing:

boot_device the boot device, one of *ironic.common.boot_devices* or None if it is unknown.

persistent Whether the boot device will persist to all future boots or not, None if it is unknown.

`get_properties()`
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

`get_sensors_data(task)`
Get sensors data.

Parameters **task** a TaskManager instance.

Raises FailedToGetSensorData when getting the sensor data fails.

Raises FailedToParseSensorData when parsing sensor data fails.

Raises InvalidParameterValue if required ipmi parameters are missing

Raises MissingParameterValue if a required parameter is missing.

Returns returns a dict of sensor data group by sensor type.

`get_supported_boot_devices(task)`
Get a list of the supported boot devices.

Parameters **task** a task from TaskManager.

Returns A list with the supported boot devices defined in *ironic.common.boot_devices*.

`inject_nmi(task)`
Inject NMI, Non Maskable Interrupt.

Inject NMI (Non Maskable Interrupt) for a node immediately.

Parameters **task** A TaskManager instance containing the node to act on.

Raises IPMIFailure on an error from ipmitool.

Returns None

`set_boot_device(task, device, persistent=False)`
Set the boot device for the tasks node.

Set the boot device to use on next reboot of the node.

Parameters

- **task** a task from TaskManager.
- **device** the boot device, one of *ironic.common.boot_devices*.
- **persistent** Boolean value. True if the boot device will persist to all future boots, False if not. Default: False.

Raises InvalidParameterValue if an invalid boot device is specified

Raises MissingParameterValue if required ipmi parameters are missing.

Raises IPMIFailure on an error from ipmitool.

validate(*task*)
Check that driver_info contains IPMI credentials.

Validates whether the driver_info property of the supplied tasks node contains the required credentials information.

Parameters **task** a task from TaskManager.

Raises InvalidParameterValue if required IPMI parameters are missing.

Raises MissingParameterValue if a required parameter is missing.

class ironic.drivers.modules.ipmitool.**IPMIPower**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.PowerInterface*

get_power_state(*task*)
Get the current power state of the tasks node.

Parameters **task** a TaskManager instance containing the node to act on.

Returns one of ironic.common.states POWER_OFF, POWER_ON or ERROR.

Raises InvalidParameterValue if required ipmi parameters are missing.

Raises MissingParameterValue if a required parameter is missing.

Raises IPMIFailure on an error from ipmitool (from _power_status call).

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

get_supported_power_states(*task*)
Get a list of the supported power states.

Parameters **task** A TaskManager instance containing the node to act on. currently not used.

Returns A list with the supported power states defined in *ironic.common.states*.

reboot(*task*, *timeout=None*)
Cycles the power to the tasks node.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **timeout** timeout (in seconds) positive integer (> 0) for any power state. The timeout is counted once during power off and once during power on for reboots. None indicates that the default timeout will be used.

Raises MissingParameterValue if required ipmi parameters are missing.

Raises InvalidParameterValue if an invalid power state was specified.

Raises PowerStateFailure if the final state of the node is not POWER_ON or the intermediate state of the node is not POWER_OFF.

set_power_state(*task*, *power_state*, *timeout=None*)
Turn the power on, off, soft reboot, or soft power off.

Parameters

- **task** a TaskManager instance containing the node to act on.
- **power_state** desired power state. one of ironic.common.states, POWER_ON, POWER_OFF, SOFT_POWER_OFF, or SOFT_REBOOT.
- **timeout** timeout (in seconds) positive integer (> 0) for any power state. The timeout is counted once during power off and once during power on for reboots. None indicates that the default timeout will be used.

Raises InvalidParameterValue if an invalid power state was specified.

Raises MissingParameterValue if required ipmi parameters are missing

Raises PowerStateFailure if the power couldnt be set to pstate.

validate(*task*)

Validate driver_info for ipmitool driver.

Check that node[driver_info] contains IPMI credentials.

Parameters **task** a TaskManager instance containing the node to act on.

Raises InvalidParameterValue if required ipmi parameters are missing.

Raises MissingParameterValue if a required parameter is missing.

class ironic.drivers.modules.ipmitool.**IPMIShellinaboxConsole**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.ipmitool.IPMIConsole*

A ConsoleInterface that uses ipmitool and shellinabox.

get_console(*task*)

Get the type and connection information about the console.

start_console(*task*)

Start a remote console for the node.

Parameters **task** a task from TaskManager

Raises InvalidParameterValue if required ipmi parameters are missing

Raises PasswordFileFailedToCreate if unable to create a file containing the password

Raises ConsoleError if the directory for the PID file cannot be created

Raises ConsoleSubprocessFailed when invoking the subprocess failed

stop_console(*task*)

Stop the remote console session for the node.

Parameters **task** a task from TaskManager

Raises ConsoleError if unable to stop the console

class ironic.drivers.modules.ipmitool.**IPMISocatConsole**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.ipmitool.IPMIConsole*

A ConsoleInterface that uses ipmitool and socat.

get_console(*task*)

Get the type and connection information about the console.

Parameters **task** a task from TaskManager

start_console(*task*)
Start a remote console for the node.

Parameters **task** a task from TaskManager

Raises InvalidParameterValue if required ipmi parameters are missing

Raises PasswordFileFailedToCreate if unable to create a file containing the password

Raises ConsoleError if the directory for the PID file cannot be created

Raises ConsoleSubprocessFailed when invoking the subprocess failed

stop_console(*task*)
Stop the remote console session for the node.

Parameters **task** a task from TaskManager

Raises ConsoleError if unable to stop the console

class ironic.drivers.modules.ipmitool.**VendorPassthru**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.VendorInterface*

bmc_reset(*task*, *http_method*, *warm=True*)
Reset BMC with IPMI command bmc reset (warm|cold).

Parameters

- **task** a TaskManager instance.
- **http_method** the HTTP method used on the request.
- **warm** boolean parameter to decide on warm or cold reset.

Raises IPMIFailure on an error from ipmitool.

Raises MissingParameterValue if a required parameter is missing.

Raises InvalidParameterValue when an invalid value is specified

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

send_raw(*task*, *http_method*, *raw_bytes*)
Send raw bytes to the BMC. Bytes should be a string of bytes.

Parameters

- **task** a TaskManager instance.
- **http_method** the HTTP method used on the request.
- **raw_bytes** a string of raw bytes to send, e.g. 0x00 0x01

Raises IPMIFailure on an error from ipmitool.

Raises MissingParameterValue if a required parameter is missing.

Raises InvalidParameterValue when an invalid value is specified.

validate(*task*, *method*, ***kwargs*)
Validate vendor-specific actions.

If invalid, raises an exception; otherwise returns None.

Valid methods:

- send_raw
- bmc_reset

Parameters

- **task** a task from TaskManager.
- **method** method to be validated
- **kwargs** info for action.

Raises InvalidParameterValue when an invalid parameter value is specified.

Raises MissingParameterValue if a required parameter is missing.

ironic.drivers.modules.ipmitool.**check_cipher_suite_errors**(*cmd_stderr*)
Checks if the command stderr contains cipher suite errors.

Parameters **cmd_stderr** The command stderr.

Returns True if the cmd_stderr contains a cipher suite error, False otherwise.

ironic.drivers.modules.ipmitool.**choose_cipher_suite**(*actual_cipher_suite*)
Gives the possible next avaible cipher suite version.

Based on CONF.ipmi.cipher_suite_versions and the last cipher suite version used that failed. This function is only called if the node doesnt have cipher_suite set. Starts using the last element of the list and decreasing the index.

Parameters **actual_cipher_suite** latest cipher suite used in the ipmi call.

Returns the next possible cipher suite or None in case of empty configuration.

ironic.drivers.modules.ipmitool.**dump_sdr**(*task*, *file_path*)
Dump SDR data to a file.

Parameters

- **task** a TaskManager instance.
- **file_path** the path to SDR dump file.

Raises IPMIFailure on an error from ipmitool.

Raises MissingParameterValue if a required parameter is missing.

Raises InvalidParameterValue when an invalid value is specified.

ironic.drivers.modules.ipmitool.**send_raw**(*task*, *raw_bytes*)
Send raw bytes to the BMC. Bytes should be a string of bytes.

Parameters

- **task** a TaskManager instance.
- **raw_bytes** a string of raw bytes to send, e.g. 0x00 0x01

Returns a tuple with stdout and stderr.

Raises IPMIFailure on an error from ipmitool.

Raises MissingParameterValue if a required parameter is missing.

Raises InvalidParameterValue when an invalid value is specified.

ironic.drivers.modules.ipmitool.**update_cipher_suite_cmd**(*actual_cs*, *args*)

Updates variables and the cipher suite cmd.

This function updates the values for all parameters so they can be used in the next retry of _exec_ipmitool.

Parameters

- **actual_cs** a string that represents the cipher suite that was used in the command.
- **args** a list that contains the ipmitool command that was executed, it will be modified in-place.

Returns the next actual_cs

ironic.drivers.modules.ipxe module

iPXE Boot Interface

class ironic.drivers.modules.ipxe.**iPXEBoot**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.pxe_base.PXEBaseMixin*, *ironic.drivers.base.BootInterface*

capabilities = ['iscsi_volume_boot', 'ramdisk_boot', 'ipxe_boot']

ipxe_enabled = True

ironic.drivers.modules.noop module

Dummy interface implementations for use as defaults with optional interfaces.

Note that unlike fake implementations, these do not pass validation and raise exceptions for user-accessible actions.

class ironic.drivers.modules.noop.**FailMixin**

Bases: object

Mixin to add to an interface to make it fail validation.

get_properties()

validate(*task*, **args*, ***kwargs*)

class ironic.drivers.modules.noop.**NoBIOS**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.noop.FailMixin*, *ironic.drivers.base.BIOSInterface*

BIOS interface implementation that raises errors on all requests.

apply_configuration(*task*, *settings*)
Validate & apply BIOS settings on the given node.

This method takes the BIOS settings from the settings param and applies BIOS settings on the given node. It may also validate the given bios settings before applying any settings and manage failures when setting an invalid BIOS config. In the case of needing password to update the BIOS config, it will be taken from the driver_info properties. After the BIOS configuration is done, cache_bios_settings will be called to update the nodes BIOS setting table with the BIOS configuration applied on the node.

Parameters

- **task** a TaskManager instance.
- **settings** Dictonary containing the BIOS configuration.

Raises UnsupportedDriverExtension, if the nodes driver doesnt support BIOS configuration.

Raises InvalidParameterValue, if validation of settings fails.

Raises MissingParameterValue, if some required parameters are missing.

Returns states.CLEANWAIT if BIOS configuration is in progress asynchronously or None if it is complete.

cache_bios_settings(*task*)
Store or update BIOS properties on the given node.

This method stores BIOS properties to the bios_settings table during cleaning operation and updates bios_settings table when apply_configuration() and factory_reset() are called to set new BIOS configurations. It will also update the timestamp of each bios setting.

Parameters **task** a TaskManager instance.

Raises UnsupportedDriverExtension, if the nodes driver doesnt support getting BIOS properties from bare metal.

Returns None.

factory_reset(*task*)
Reset BIOS configuration to factory default on the given node.

This method resets BIOS configuration to factory default on the given node. After the BIOS reset action is done, cache_bios_settings will be called to update the nodes BIOS settings table with default bios settings.

Parameters **task** a TaskManager instance.

Raises UnsupportedDriverExtension, if the nodes driver doesnt support BIOS reset.

Returns states.CLEANWAIT if BIOS configuration is in progress asynchronously or None if it is complete.

class ironic.drivers.modules.noop.**NoConsole**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.noop.FailMixin*, *ironic.drivers.base.ConsoleInterface*

Console interface implementation that raises errors on all requests.

get_console(*task*, **args*, ***kwargs*)
Get connection information about the console.

This method should return the necessary information for the client to access the console.

Parameters **task** A TaskManager instance containing the node to act on.

Returns the console connection information.

start_console(*task*, **args*, ***kwargs*)
Start a remote console for the tasks node.

This method should not raise an exception if console already started.

Parameters **task** A TaskManager instance containing the node to act on.

stop_console(*task*, **args*, ***kwargs*)
Stop the remote console session for the tasks node.

Parameters **task** A TaskManager instance containing the node to act on.

class ironic.drivers.modules.noop.**NoInspect**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.noop.FailMixin*, *ironic.drivers.base.InspectInterface*

Inspect interface implementation that raises errors on all requests.

inspect_hardware(*task*, **args*, ***kwargs*)
Inspect hardware.

Inspect hardware to obtain the essential & additional hardware properties.

Parameters **task** A task from TaskManager.

Raises HardwareInspectionFailure, if unable to get essential hardware properties.

Returns Resulting state of the inspection i.e. states.MANAGEABLE or None.

class ironic.drivers.modules.noop.**NoRAID**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.noop.FailMixin*, *ironic.drivers.base.RAIDInterface*

RAID interface implementation that raises errors on all requests.

create_configuration(*task*, **args*, ***kwargs*)
Creates RAID configuration on the given node.

This method creates a RAID configuration on the given node. It assumes that the target RAID configuration is already available in node.target_raid_config. Implementations of this interface are supposed to read the RAID configuration from node.target_raid_config. After the RAID configuration is done (either in this method OR in a call-back method), ironic.common.raid.update_raid_info() may be called to sync the nodes RAID-related information with the RAID configuration applied on the node.

Parameters

- **task** A TaskManager instance.
- **create_root_volume** Setting this to False indicates not to create root volume that is specified in the nodes target_raid_config. Default value is True.

- **create_nonroot_volumes** Setting this to False indicates not to create non-root volumes (all except the root volume) in the nodes target_raid_config. Default value is True.
- **delete_existing** Setting this to True indicates to delete RAID configuration prior to creating the new configuration.

Returns states.CLEANWAIT (cleaning) or states.DEPLOYWAIT (deployment) if RAID configuration is in progress asynchronously, or None if it is complete.

delete_configuration(*task*, **args*, ***kwargs*)
Deletes RAID configuration on the given node.

This method deletes the RAID configuration on the give node. After RAID configuration is deleted, node.raid_config should be cleared by the implementation.

Parameters **task** A TaskManager instance.

Returns states.CLEANWAIT (cleaning) or states.DEPLOYWAIT (deployment) if deletion is in progress asynchronously, or None if it is complete.

validate_raid_config(*task*, *raid_config*)
Validates the given RAID configuration.

This method validates the given RAID configuration. Driver implementations of this interface can override this method to support custom parameters for RAID configuration.

Parameters

- **task** A TaskManager instance.
- **raid_config** The RAID configuration to validate.

Raises InvalidParameterValue, if the RAID configuration is invalid.

class ironic.drivers.modules.noop.**NoRescue**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.noop.FailMixin*, *ironic.drivers.base.RescueInterface*

Rescue interface implementation that raises errors on all requests.

rescue(*task*, **args*, ***kwargs*)
Boot the tasks node into a rescue environment.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InstanceRescueFailure if node validation or rescue operation fails.

Returns states.RESCUEWAIT if rescue is in progress asynchronously or states.RESCUE if it is complete.

unrescue(*task*, **args*, ***kwargs*)
Tear down the rescue environment, and return to normal.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InstanceUnrescueFailure if node validation or unrescue operation fails.

Returns states.ACTIVE if it is successful.

class ironic.drivers.modules.noop.**NoVendor**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.noop.FailMixin*, *ironic.drivers.base.VendorInterface*

Vendor interface implementation that raises errors on all requests.

driver_validate(*method*, ***kwargs*)
Validate driver-vendor-passthru actions.

If invalid, raises an exception; otherwise returns None.

Parameters

- **method** method to be validated
- **kwargs** info for action.

Raises MissingParameterValue if kwargs does not contain certain parameter.

Raises InvalidParameterValue if parameter does not match.

ironic.drivers.modules.noop_mgmt module

No-op management interface implementation.

class ironic.drivers.modules.noop_mgmt.**NoopManagement**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.ManagementInterface*

No-op management interface implementation.

Using this implementation requires the boot order to be preconfigured to first try PXE booting, then fall back to hard drives.

get_boot_device(*task*)
Get the current boot device for a node.

Provides the current boot device of the node. Be aware that not all drivers support this.

Parameters **task** A task from TaskManager.

Raises MissingParameterValue if a required parameter is missing

Returns

A dictionary containing:

boot_device Ahe boot device, one of *ironic.common.boot_devices* or None if it is unknown.

persistent Whether the boot device will persist to all future boots or not, None if it is unknown.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

get_sensors_data(*task*)
Get sensors data method.

Parameters **task** A TaskManager instance.

Raises FailedToGetSensorData when getting the sensor data fails.

Raises FailedToParseSensorData when parsing sensor data fails.

Returns

Returns a consistent format dict of sensor data grouped by sensor type, which can be processed by Ceilometer. eg,

```
{
  'Sensor Type 1': {
    'Sensor ID 1': {
      'Sensor Reading': 'current value',
      'key1': 'value1',
      'key2': 'value2'
    },
    'Sensor ID 2': {
      'Sensor Reading': 'current value',
      'key1': 'value1',
      'key2': 'value2'
    }
  },
  'Sensor Type 2': {
    'Sensor ID 3': {
      'Sensor Reading': 'current value',
      'key1': 'value1',
      'key2': 'value2'
    },
    'Sensor ID 4': {
      'Sensor Reading': 'current value',
      'key1': 'value1',
      'key2': 'value2'
    }
  }
}
```

get_supported_boot_devices(*task*)

Get a list of the supported boot devices.

Parameters **task** A task from TaskManager.

Returns A list with the supported boot devices defined in *ironic.common.boot_devices*.

set_boot_device(*task*, *device*, *persistent=False*)

Set the boot device for a node.

Set the boot device to use on next reboot of the node.

Parameters

- **task** A task from TaskManager.
- **device** The boot device, one of *ironic.common.boot_devices*.
- **persistent** Boolean value. True if the boot device will persist to all future boots, False if not. Default: False.

Raises InvalidParameterValue if an invalid boot device is specified.

Raises MissingParameterValue if a required parameter is missing

validate(*task*)
Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

ironic.drivers.modules.pxe module

PXE Boot Interface

class ironic.drivers.modules.pxe.**PXEAnacondaDeploy**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.agent_base.AgentBaseMixin*, *ironic.drivers.modules.agent_base.HeartbeatMixin*, *ironic.drivers.base.DeployInterface*

clean_up(*task*)
Clean up the deployment environment for the tasks node.

Unlinks TFTP and instance images and triggers image cache cleanup. Removes the TFTP configuration files for this node.

Parameters **task** a TaskManager instance containing the node to act on.

deploy(*task*)
Perform a deployment to the tasks node.

Perform the necessary work to deploy an image onto the specified node. This method will be called after prepare(), which may have already performed any preparatory steps, such as pre-caching some data for the node.

Parameters **task** A TaskManager instance containing the node to act on.

Returns status of the deploy. One of ironic.common.states.

deploy_has_started(*task*)

deploy_is_done(*task*)

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

prepare(*task*)
Prepare the deployment environment for the tasks node.

If preparation of the deployment environment ahead of time is possible, this method should be implemented by the driver.

If implemented, this method must be idempotent. It may be called multiple times for the same node on the same conductor.

This method is called before *deploy*.

Parameters **task** A TaskManager instance containing the node to act on.

reboot_to_instance(*task*)
Method invoked after the deployment is completed.

Parameters **task** a TaskManager instance

should_manage_boot(*task*)
Whether agent boot is managed by ironic.

validate(*task*)
Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

class ironic.drivers.modules.pxe.**PXEBoot**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.pxe_base.PXEBaseMixin*, *ironic.drivers.base.BootInterface*

capabilities = ['ramdisk_boot', 'pxe_boot']

ironic.drivers.modules.pxe_base module

Base PXE Interface Methods

class ironic.drivers.modules.pxe_base.**PXEBaseMixin**
Bases: object

clean_up_instance(*task*)
Cleans up the boot of instance.

This method cleans up the environment that was setup for booting the instance. It unlinks the instance kernel/ramdisk in nodes directory in tftproot and removes the PXE config.

Parameters **task** a task from TaskManager.

Returns None

clean_up_ramdisk(*task*)
Cleans up the boot of ironic ramdisk.

This method cleans up the PXE environment that was setup for booting the deploy or rescue ramdisk. It unlinks the deploy/rescue kernel/ramdisk in the nodes directory in tftproot and removes its PXE config.

Parameters

- **task** a task from TaskManager.

- **mode** Label indicating a deploy or rescue operation was carried out on the node. Supported values are deploy and rescue. Defaults to deploy, indicating deploy operation was carried out.

Returns None

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

ipxe_enabled = False

prepare_instance(*task*)
Prepares the boot of instance.

This method prepares the boot of the instance after reading relevant information from the nodes instance_info. In case of netboot, it updates the dhcp entries and switches the PXE config. In case of localboot, it cleans up the PXE config.

Parameters **task** a task from TaskManager.

Returns None

prepare_ramdisk(*task*, *ramdisk_params*)
Prepares the boot of Ironic ramdisk using PXE.

This method prepares the boot of the deploy or rescue kernel/ramdisk after reading relevant information from the nodes driver_info and instance_info.

Parameters

- **task** a task from TaskManager.
- **ramdisk_params** the parameters to be passed to the ramdisk. pxe driver passes these parameters as kernel command-line arguments.

Returns None

Raises MissingParameterValue, if some information is missing in nodes driver_info or instance_info.

Raises InvalidParameterValue, if some information provided is invalid.

Raises IronicException, if some power or set boot boot device operation failed on the node.

validate(*task*)
Validate the PXE-specific info for booting deploy/instance images.

This method validates the PXE-specific info for booting the ramdisk and instance on the node. If invalid, raises an exception; otherwise returns None.

Parameters **task** a task from TaskManager.

Returns None

Raises InvalidParameterValue, if some parameters are invalid.

Raises MissingParameterValue, if some required parameters are missing.

validate_inspection(*task*)
Validate that the node has required properties for inspection.

Parameters **task** A TaskManager instance with the node being checked

Raises UnsupportedDriverExtension

validate_rescue(*task*)
Validate that the node has required properties for rescue.

Parameters **task** a TaskManager instance with the node being checked

Raises MissingParameterValue if node is missing one or more required parameters

ironic.drivers.modules.ramdisk module

Ramdisk Deploy Interface

class ironic.drivers.modules.ramdisk.**RamdiskDeploy**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.agent_base.AgentBaseMixin*, *ironic.drivers.modules.agent_base.HeartbeatMixin*, *ironic.drivers.base.DeployInterface*

deploy(*task*)
Perform a deployment to the tasks node.

Perform the necessary work to deploy an image onto the specified node. This method will be called after prepare(), which may have already performed any preparatory steps, such as pre-caching some data for the node.

Parameters **task** A TaskManager instance containing the node to act on.

Returns status of the deploy. One of ironic.common.states.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

prepare(*task*)
Prepare the deployment environment for the tasks node.

If preparation of the deployment environment ahead of time is possible, this method should be implemented by the driver.

If implemented, this method must be idempotent. It may be called multiple times for the same node on the same conductor.

This method is called before *deploy*.

Parameters **task** A TaskManager instance containing the node to act on.

validate(*task*)
Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

ironic.drivers.modules.snmp module

Ironic SNMP power manager.

Provides basic power control using an SNMP-enabled smart power controller. Uses a pluggable driver model to support devices with different SNMP object models.

class ironic.drivers.modules.snmp.**SNMPClient**(*address*, *port*, *version*, *read_community=None*, *write_community=None*, *user=None*, *auth_proto=None*, *auth_key=None*, *priv_proto=None*, *priv_key=None*, *context_engine_id=None*, *context_name=None*)

Bases: `object`

SNMP client object.

Performs low level SNMP get and set operations. Encapsulates all interaction with PySNMP to simplify dynamic importing and unit testing.

get(*oid*)

Use PySNMP to perform an SNMP GET operation on a single object.

Parameters **oid** The OID of the object to get.

Raises SNMPFailure if an SNMP request fails.

Returns The value of the requested object.

get_next(*oid*)

Use PySNMP to perform an SNMP GET NEXT operation on a table object.

Parameters **oid** The OID of the object to get.

Raises SNMPFailure if an SNMP request fails.

Returns A list of values of the requested table object.

set(*oid*, *value*)

Use PySNMP to perform an SNMP SET operation on a single object.

Parameters

- **oid** The OID of the object to set.
- **value** The value of the object to set.

Raises SNMPFailure if an SNMP request fails.

class ironic.drivers.modules.snmp.**SNMPDriverAPCMasterSwitch**(**args*, ***kwargs*)

Bases: *ironic.drivers.modules.snmp.SNMPDriverSimple*

SNMP driver class for APC MasterSwitch PDU devices.

SNMP objects for APC SNMPDriverAPCMasterSwitch PDU: 1.3.6.1.4.1.318.1.1.4.4.2.1.3 sPDUOutletCtl Values: 1=On, 2=Off, 3=PowerCycle, [more options follow]

oid_device = (318, 1, 1, 4, 4, 2, 1, 3)

system_id = (318, 1, 1, 4)

value_power_off = 2

value_power_on = 1

class ironic.drivers.modules.snmp.**SNMPDriverAPCMasterSwitchPlus**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.snmp.SNMPDriverSimple*

SNMP driver class for APC MasterSwitchPlus PDU devices.

SNMP objects for APC SNMPDriverAPCMasterSwitchPlus PDU: 1.3.6.1.4.1.318.1.1.6.5.1.1.5 sPDUOutletControlMSPOutletCommand Values: 1=On, 3=Off, [more options follow]

oid_device = (318, 1, 1, 6, 5, 1, 1, 5)

system_id = (318, 1, 1, 6)

value_power_off = 3

value_power_on = 1

class ironic.drivers.modules.snmp.**SNMPDriverAPCRackPDU**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.snmp.SNMPDriverSimple*

SNMP driver class for APC RackPDU devices.

SNMP objects for APC SNMPDriverAPCRackPDU PDU: # 1.3.6.1.4.1.318.1.1.12.3.3.1.1.4 rPDUOutletControlOutletCommand Values: 1=On, 2=Off, 3=PowerCycle, [more options follow]

oid_device = (318, 1, 1, 12, 3, 3, 1, 1, 4)

system_id = (318, 1, 1, 12)

value_power_off = 2

value_power_on = 1

class ironic.drivers.modules.snmp.**SNMPDriverAten**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.snmp.SNMPDriverSimple*

SNMP driver class for Aten PDU devices.

SNMP objects for Aten PDU: 1.3.6.1.4.1.21317.1.3.2.2.2.2 Outlet Power Values: 1=Off, 2=On, 3=Pending, 4=Reset

oid_device = (21317, 1, 3, 2, 2, 2, 2)

system_id = (21317,)

value_power_off = 1

value_power_on = 2

class ironic.drivers.modules.snmp.**SNMPDriverAuto**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.snmp.SNMPDriverBase*

SYS_OBJ_OID = (1, 3, 6, 1, 2, 1, 1, 2)

class ironic.drivers.modules.snmp.**SNMPDriverBase**(*snmp_info*)
Bases: object

SNMP power driver base class.

The SNMPDriver class hierarchy implements manufacturer-specific MIB actions over SNMP to interface with different smart power controller products.

oid_enterprise = (1, 3, 6, 1, 4, 1)

power_off()
Set the power state to this node to OFF.

Raises SNMPFailure if an SNMP request fails.

Returns power state. One of *ironic.common.states*.

power_on()
Set the power state to this node to ON.

Raises SNMPFailure if an SNMP request fails.

Returns power state. One of *ironic.common.states*.

power_reset()
Reset the power to this node.

Raises SNMPFailure if an SNMP request fails.

Returns power state. One of *ironic.common.states*.

power_state()
Returns a nodes current power state.

Raises SNMPFailure if an SNMP request fails.

Returns power state. One of *ironic.common.states*.

retry_interval = 1

class ironic.drivers.modules.snmp.**SNMPDriverBaytechMRP27**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.snmp.SNMPDriverSimple*

SNMP driver class for Baytech MRP27 PDU devices.

SNMP objects for Baytech MRP27 PDU: 4779, 1, 3, 5, 3, 1, 3, {unit_id} Outlet Power Values: 0=Off, 1=On, 2=Reboot

oid_device = (4779, 1, 3, 5, 3, 1, 3, 1)

unit_id = 1

value_power_off = 0

value_power_on = 1

class ironic.drivers.modules.snmp.**SNMPDriverCyberPower**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.snmp.SNMPDriverSimple*

SNMP driver class for CyberPower PDU devices.

SNMP objects for CyberPower PDU: 1.3.6.1.4.1.3808.1.1.3.3.3.1.1.4 ePDUOutletControlOutletCommand Values: 1=On, 2=Off, 3=PowerCycle, [more options follow]

oid_device = (3808, 1, 1, 3, 3, 3, 1, 1, 4)

system_id = (3808,)

value_power_off = 2

value_power_on = 1

class ironic.drivers.modules.snmp.**SNMPDriverEatonPower**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.snmp.SNMPDriverBase*

SNMP driver class for Eaton Power PDU.

The Eaton power PDU does not follow the model of SNMPDriverSimple as it uses multiple SNMP objects.

SNMP objects for Eaton Power PDU 1.3.6.1.4.1.534.6.6.7.6.6.1.2.<outlet ID> outletControlStatus Read 0=off, 1=on, 2=pending off, 3=pending on 1.3.6.1.4.1.534.6.6.7.6.6.1.3.<outlet ID> outletControlOffCmd Write 0 for immediate power off 1.3.6.1.4.1.534.6.6.7.6.6.1.4.<outlet ID> outletControlOnCmd Write 0 for immediate power on

oid_device = (534, 6, 6, 7, 6, 6, 1)

oid_poweroff = (4,)

oid_poweron = (3,)

oid_status = (2,)

status_off = 0

status_on = 1

status_pending_off = 2

status_pending_on = 3

system_id = (534,)

value_power_off = 0

value_power_on = 0

class ironic.drivers.modules.snmp.**SNMPDriverSimple**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.snmp.SNMPDriverBase*

SNMP driver base class for simple PDU devices.

Here, simple refers to devices which provide a single SNMP object for controlling the power state of an outlet.

The default OID of the power state object is of the form <enterprise OID>.<device OID>.<outlet ID>. A different OID may be specified by overriding the _snmp_oid method in a subclass.

abstract property oid_device
Device dependent portion of the power state object OID.

abstract property value_power_off
Value representing power off state.

abstract property value_power_on
Value representing power on state.

class ironic.drivers.modules.snmp.**SNMPDriverTeltronix**(**args*, ***kwargs*)
Bases: *ironic.drivers.modules.snmp.SNMPDriverSimple*

SNMP driver class for Teltronix PDU devices.

SNMP objects for Teltronix PDU: 1.3.6.1.4.1.23620.1.2.2.1.4 Outlet Power Values: 1=Off, 2=On

oid_device = (23620, 1, 2, 2, 1, 4)

system_id = (23620,)

value_power_off = 1

value_power_on = 2

class ironic.drivers.modules.snmp.**SNMPPower**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.PowerInterface*

SNMP Power Interface.

This PowerInterface class provides a mechanism for controlling the power state of a physical device using an SNMP-enabled smart power controller.

get_power_state(*task*)

Get the current power state.

Poll the SNMP device for the current power state of the node.

Parameters **task** An instance of *ironic.manager.task_manager.TaskManager*.

Raises MissingParameterValue if required SNMP parameters are missing.

Raises InvalidParameterValue if SNMP parameters are invalid.

Raises SNMPFailure if an SNMP request fails.

Returns power state. One of *ironic.common.states*.

get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

reboot(*task*, *timeout=None*)

Cycles the power to a node.

Parameters

- **task** An instance of *ironic.manager.task_manager.TaskManager*.
- **timeout** timeout (in seconds). Unsupported by this interface.

Raises MissingParameterValue if required SNMP parameters are missing.

Raises InvalidParameterValue if SNMP parameters are invalid.

Raises PowerStateFailure if the final power state of the node is not POWER_ON after the timeout.

Raises SNMPFailure if an SNMP request fails.

set_power_state(*task*, *pstate*, *timeout=None*)

Turn the power on or off.

Set the power state of a node.

Parameters

- **task** An instance of *ironic.manager.task_manager.TaskManager*.
- **pstate** Either POWER_ON or POWER_OFF from :class: *ironic.common.states*.

- **timeout** timeout (in seconds). Unsupported by this interface.

Raises MissingParameterValue if required SNMP parameters are missing.

Raises InvalidParameterValue if SNMP parameters are invalid or *pstate* is invalid.

Raises PowerStateFailure if the final power state of the node is not as requested after the timeout.

Raises SNMPFailure if an SNMP request fails.

validate(*task*)
Check that node.driver_info contains the requisite fields.

Raises MissingParameterValue if required SNMP parameters are missing.

Raises InvalidParameterValue if SNMP parameters are invalid.

ironic.drivers.modules.snmp.**memoize**(*f*)

ironic.drivers.modules.snmp.**retry_on_outdated_cache**(*f*)

Module contents

Submodules

ironic.drivers.base module

Abstract base classes for drivers.

ironic.drivers.base.**ALL_INTERFACES = {'bios', 'boot', 'console', 'deploy', 'inspect', 'management', 'network', 'power', 'raid', 'rescue', 'storage', 'vendor'}**
Constant holding all known interfaces.

class ironic.drivers.base.**BIOSInterface**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.BaseInterface*

abstract apply_configuration(*task*, *settings*)
Validate & apply BIOS settings on the given node.

This method takes the BIOS settings from the settings param and applies BIOS settings on the given node. It may also validate the given bios settings before applying any settings and manage failures when setting an invalid BIOS config. In the case of needing password to update the BIOS config, it will be taken from the driver_info properties. After the BIOS configuration is done, cache_bios_settings will be called to update the nodes BIOS setting table with the BIOS configuration applied on the node.

Parameters

- **task** a TaskManager instance.
- **settings** Dictonary containing the BIOS configuration.

Raises UnsupportedDriverExtension, if the nodes driver doesnt support BIOS configuration.

Raises InvalidParameterValue, if validation of settings fails.

Raises MissingParameterValue, if some required parameters are missing.

Returns states.CLEANWAIT if BIOS configuration is in progress asynchronously or None if it is complete.

abstract cache_bios_settings(*task*)

Store or update BIOS properties on the given node.

This method stores BIOS properties to the bios_settings table during cleaning operation and updates bios_settings table when apply_configuration() and factory_reset() are called to set new BIOS configurations. It will also update the timestamp of each bios setting.

Parameters **task** a TaskManager instance.

Raises UnsupportedDriverExtension, if the nodes driver doesnt support getting BIOS properties from bare metal.

Returns None.

abstract factory_reset(*task*)

Reset BIOS configuration to factory default on the given node.

This method resets BIOS configuration to factory default on the given node. After the BIOS reset action is done, cache_bios_settings will be called to update the nodes BIOS settings table with default bios settings.

Parameters **task** a TaskManager instance.

Raises UnsupportedDriverExtension, if the nodes driver doesnt support BIOS reset.

Returns states.CLEANWAIT if BIOS configuration is in progress asynchronously or None if it is complete.

interface_type = 'bios'

Interface type, used for clean steps and logging.

class ironic.drivers.base.**BareDriver**

Bases: object

A bare driver object which will have interfaces attached later.

Any composable interfaces should be added as class attributes of this class, as well as appended to core_interfaces or standard_interfaces here.

property all_interfaces

bios = None

Standard attribute for BIOS related features.

A reference to an instance of :class:BIOSInterface.

boot = None

Standard attribute for boot related features.

A reference to an instance of :class:BootInterface.

console = None

Standard attribute for managing console access.

A reference to an instance of :class:ConsoleInterface.

property core_interfaces
Interfaces that are required to be implemented.

deploy = None
Core attribute for managing deployments.

A reference to an instance of :class:DeployInterface.

get_properties()
Get the properties of the driver.

Returns dictionary of <property name>:<property description> entries.

inspect = None
Standard attribute for inspection related features.

A reference to an instance of :class:InspectInterface.

management = None
Standard attribute for management related features.

A reference to an instance of :class:ManagementInterface.

network = None
Core attribute for network connectivity.

A reference to an instance of :class:NetworkInterface.

property non_vendor_interfaces

property optional_interfaces
Interfaces that can be no-op.

power = None
Core attribute for managing power state.

A reference to an instance of :class:PowerInterface.

raid = None
Standard attribute for RAID related features.

A reference to an instance of :class:RaidInterface.

rescue = None
Standard attribute for accessing rescue features.

A reference to an instance of :class:RescueInterface.

storage = None
Standard attribute for (remote) storage interface.

A reference to an instance of :class:StorageInterface.

vendor = None
Attribute for accessing any vendor-specific extensions.

A reference to an instance of :class:VendorInterface.

class ironic.drivers.base.**BaseInterface**(**args*, ***kwargs*)
Bases: object

A base interface implementing common functions for Driver Interfaces.

execute_clean_step(*task*, *step*)
Execute the clean step on task.node.

A clean step must take a single positional argument: a TaskManager object. It may take one or more keyword variable arguments (for use with manual cleaning only.)

A step can be executed synchronously or asynchronously. A step should return None if the method has completed synchronously or states.CLEANWAIT if the step will continue to execute asynchronously. If the step executes asynchronously, it should issue a call to the continue_node_clean RPC, so the conductor can begin the next clean step.

Parameters

- **task** A TaskManager object
- **step** The clean step dictionary representing the step to execute

Returns None if this method has completed synchronously, or states.CLEANWAIT if the step will continue to execute asynchronously.

execute_deploy_step(*task*, *step*)
Execute the deploy step on task.node.

A deploy step must take a single positional argument: a TaskManager object. It may take one or more keyword variable arguments (for use in the future, when deploy steps can be specified via the API).

A step can be executed synchronously or asynchronously. A step should return None if the method has completed synchronously or states.DEPLOYWAIT if the step will continue to execute asynchronously. If the step executes asynchronously, it should issue a call to the continue_node_deploy RPC, so the conductor can begin the next deploy step.

Parameters

- **task** A TaskManager object
- **step** The deploy step dictionary representing the step to execute

Returns None if this method has completed synchronously, or states.DEPLOYWAIT if the step will continue to execute asynchronously.

execute_verify_step(*task*, *step*)
Execute the verify step on task.node.

A verify step must take a single positional argument: a TaskManager object. It does not take keyword variable arguments.

Parameters

- **task** A TaskManager object
- **step** The deploy step dictionary representing the step to execute

Returns None if this method has completed synchronously

get_clean_steps(*task*)
Get a list of (enabled and disabled) clean steps for the interface.

This function will return all clean steps (both enabled and disabled) for the interface, in an unordered list.

Parameters **task** A TaskManager object, useful for interfaces overriding this function

Raises *NodeCleaningFailure* if there is a problem getting the steps from the driver. For example, when a node (using an agent driver) has just been enrolled and the agent isnt alive yet to be queried for the available clean steps.

Returns A list of clean step dictionaries

get_deploy_steps(*task*)

Get a list of (enabled and disabled) deploy steps for the interface.

This function will return all deploy steps (both enabled and disabled) for the interface, in an unordered list.

Parameters **task** A TaskManager object, useful for interfaces overriding this function

Raises *InstanceDeployFailure* if there is a problem getting the steps from the driver. For example, when a node (using an agent driver) has just been enrolled and the agent isnt alive yet to be queried for the available deploy steps.

Returns A list of deploy step dictionaries

abstract get_properties()

Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

get_verify_steps(*task*)

Get a list of (enabled and disabled) verify steps for the interface.

This function will return all verify steps (both enabled and disabled) for the interface, in an unordered list.

Parameters **task** A TaskManager object, useful for interfaces overriding this function

Raises *NodeVerifyFailure* if there is a problem getting the steps from the driver. For example, when a node (using an agent driver) has just been enrolled and the agent isnt alive yet to be queried for the available verify steps.

Returns A list of deploy step dictionaries

interface_type = 'base'

Interface type, used for clean steps and logging.

supported = True

Indicates if an interface is supported.

This will be set to False for interfaces which are untested in first- or third-party CI, or in the process of being deprecated.

abstract validate(*task*)

Validate the driver-specific Node deployment info.

This method validates whether the driver_info and/or instance_info properties of the tasks node contains the required information for this interface to function.

This method is often executed synchronously in API requests, so it should not conduct long-running checks.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InvalidParameterValue on malformed parameter(s)

Raises MissingParameterValue on missing parameter(s)

class ironic.drivers.base.**BootInterface**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.BaseInterface*

Interface for boot-related actions.

capabilities = []

abstract clean_up_instance(*task*)

Cleans up the boot of instance.

This method cleans up the environment that was setup for booting the instance.

Parameters **task** A task from TaskManager.

Returns None

abstract clean_up_ramdisk(*task*)

Cleans up the boot of ironic ramdisk.

This method cleans up the environment that was setup for booting the deploy or rescue ramdisk.

Parameters **task** A task from TaskManager.

Returns None

interface_type = 'boot'

Interface type, used for clean steps and logging.

abstract prepare_instance(*task*)

Prepares the boot of instance.

This method prepares the boot of the instance after reading relevant information from the nodes database.

Parameters **task** A task from TaskManager.

Returns None

abstract prepare_ramdisk(*task*, *ramdisk_params*)

Prepares the boot of Ironic ramdisk.

This method prepares the boot of the deploy or rescue ramdisk after reading relevant information from the nodes database.

Parameters

- **task** A task from TaskManager.
- **ramdisk_params** The options to be passed to the ironic ramdisk. Different implementations might want to boot the ramdisk in different ways by passing parameters to them. For example,

  When Agent ramdisk is booted to deploy a node, it takes the parameters ipa-api-url, etc.

Other implementations can make use of ramdisk_params to pass such information. Different implementations of boot interface will have different ways of passing parameters to the ramdisk.

Returns None

validate_inspection(*task*)
Validate that the node has required properties for inspection.

Parameters **task** A TaskManager instance with the node being checked

Raises MissingParameterValue if node is missing one or more required parameters

Raises UnsupportedDriverExtension

validate_rescue(*task*)
Validate that the node has required properties for rescue.

Parameters **task** A TaskManager instance with the node being checked

Raises MissingParameterValue if node is missing one or more required parameters

Raises UnsupportedDriverExtension

class ironic.drivers.base.**ConsoleInterface**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.BaseInterface*

Interface for console-related actions.

abstract get_console(*task*)
Get connection information about the console.

This method should return the necessary information for the client to access the console.

Parameters **task** A TaskManager instance containing the node to act on.

Returns the console connection information.

interface_type = 'console'
Interface type, used for clean steps and logging.

abstract start_console(*task*)
Start a remote console for the tasks node.

This method should not raise an exception if console already started.

Parameters **task** A TaskManager instance containing the node to act on.

abstract stop_console(*task*)
Stop the remote console session for the tasks node.

Parameters **task** A TaskManager instance containing the node to act on.

class ironic.drivers.base.**DeployInterface**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.BaseInterface*

Interface for deploy-related actions.

abstract clean_up(*task*)
Clean up the deployment environment for the tasks node.

If preparation of the deployment environment ahead of time is possible, this method should be implemented by the driver. It should erase anything cached by the *prepare* method.

If implemented, this method must be idempotent. It may be called multiple times for the same node on the same conductor, and it may be called by multiple conductors in parallel. Therefore, it must not require an exclusive lock.

This method is called before *tear_down*.

Parameters `task` A TaskManager instance containing the node to act on.

`abstract deploy`(*task*)

Perform a deployment to the tasks node.

Perform the necessary work to deploy an image onto the specified node. This method will be called after prepare(), which may have already performed any preparatory steps, such as pre-caching some data for the node.

Parameters `task` A TaskManager instance containing the node to act on.

Returns status of the deploy. One of ironic.common.states.

`heartbeat`(*task*, *callback_url*, *agent_version*, *agent_verify_ca=None*, *agent_status=None*, *agent_status_message=None*)

Record a heartbeat for the node.

Parameters

- `task` A TaskManager instance containing the node to act on.
- `callback_url` a URL to use to call to the ramdisk.
- `agent_version` The version of the agent that is heartbeating
- `agent_verify_ca` TLS certificate for the agent.
- `agent_status` Status of the heartbeating agent
- `agent_status_message` Message describing the agent status

Returns None

`interface_type = 'deploy'`

Interface type, used for clean steps and logging.

`abstract prepare`(*task*)

Prepare the deployment environment for the tasks node.

If preparation of the deployment environment ahead of time is possible, this method should be implemented by the driver.

If implemented, this method must be idempotent. It may be called multiple times for the same node on the same conductor.

This method is called before *deploy*.

Parameters `task` A TaskManager instance containing the node to act on.

`prepare_cleaning`(*task*)

Prepare the node for cleaning tasks.

For example, nodes that use the Ironic Python Agent will need to boot the ramdisk in order to do in-band cleaning tasks.

If the function is asynchronous, the driver will need to handle settings node.driver_internal_info[clean_steps] and node.clean_step, as they would be set in

ironic.conductor.manager._do_node_clean, but cannot be set when this is asynchronous. After, the interface should make an RPC call to continue_node_cleaning to start cleaning.

NOTE(JoshNang) this should be moved to BootInterface when it gets implemented.

Parameters `task` A TaskManager instance containing the node to act on.

Returns If this function is going to be asynchronous, should return *states.CLEANWAIT*. Otherwise, should return *None*. The interface will need to call _get_cleaning_steps and then RPC to continue_node_cleaning

`abstract take_over`(*task*)

Take over management of this tasks node from a dead conductor.

If conductors hosts maintain a static relationship to nodes, this method should be implemented by the driver to allow conductors to perform the necessary work during the remapping of nodes to conductors when a conductor joins or leaves the cluster.

For example, the PXE driver has an external dependency: Neutron must forward DHCP BOOT requests to a conductor which has prepared the tftpboot environment for the given node. When a conductor goes offline, another conductor must change this setting in Neutron as part of remapping that nodes control to itself. This is performed within the *takeover* method.

Parameters `task` A TaskManager instance containing the node to act on.

`abstract tear_down`(*task*)

Tear down a previous deployment on the tasks node.

Given a node that has been previously deployed to, do all cleanup and tear down necessary to un-deploy that node.

Parameters `task` A TaskManager instance containing the node to act on.

Returns status of the deploy. One of ironic.common.states.

`tear_down_cleaning`(*task*)

Tear down after cleaning is completed.

Given that cleaning is complete, do all cleanup and tear down necessary to allow the node to be deployed to again.

NOTE(JoshNang) this should be moved to BootInterface when it gets implemented.

Parameters `task` A TaskManager instance containing the node to act on.

`class ironic.drivers.base.InspectInterface`(**args*, ***kwargs*)

Bases: *ironic.drivers.base.BaseInterface*

Interface for inspection-related actions.

`ESSENTIAL_PROPERTIES = {'cpu_arch', 'cpus', 'local_gb', 'memory_mb'}`

The properties required by scheduler/deploy.

`abort`(*task*)

Abort asynchronized hardware inspection.

Abort an ongoing hardware introspection, this is only used for asynchronize based inspect interface.

NOTE: This interface is called with node exclusive lock held, the interface implementation is expected to be a quick processing.

Parameters `task` a task from TaskManager.

Raises UnsupportedDriverExtension, if the method is not implemented by specific inspect interface.

`abstract inspect_hardware(task)`

Inspect hardware.

Inspect hardware to obtain the essential & additional hardware properties.

Parameters `task` A task from TaskManager.

Raises HardwareInspectionFailure, if unable to get essential hardware properties.

Returns Resulting state of the inspection i.e. states.MANAGEABLE or None.

`interface_type = 'inspect'`

Interface type, used for clean steps and logging.

`class ironic.drivers.base.ManagementInterface(*args, **kwargs)`

Bases: *ironic.drivers.base.BaseInterface*

Interface for management related actions.

`detect_vendor(task)`

Detects, stores, and returns the hardware vendor.

If the Node object `properties` field does not already contain a `vendor` field, then this method is intended to query Detects the BMC hardware vendor and stores the returned value with-in the Node object `properties` field if detected.

Parameters `task` A task from TaskManager.

Raises InvalidParameterValue if an invalid component, indicator or state is specified.

Raises MissingParameterValue if a required parameter is missing

Returns String representing the BMC reported Vendor or Manufacturer, otherwise returns None.

`abstract get_boot_device(task)`

Get the current boot device for a node.

Provides the current boot device of the node. Be aware that not all drivers support this.

Parameters `task` A task from TaskManager.

Raises MissingParameterValue if a required parameter is missing

Returns

A dictionary containing:

boot_device Ahe boot device, one of *ironic.common.boot_devices* or None if it is unknown.

persistent Whether the boot device will persist to all future boots or not, None if it is unknown.

`get_boot_mode`(*task*)
Get the current boot mode for a node.

Provides the current boot mode of the node.

NOTE: Not all drivers support this method. Older hardware may not implement that.

Parameters `task` A task from TaskManager.

Raises MissingParameterValue if a required parameter is missing

Raises DriverOperationError or its derivative in case of driver runtime error.

Raises UnsupportedDriverExtension if requested operation is not supported by the driver

Returns The boot mode, one of `ironic.common.boot_mode` or None if it is unknown.

`get_indicator_state`(*task*, *component*, *indicator*)
Get current state of the indicator of the hardware component.

Parameters

- `task` A task from TaskManager.
- `component` The hardware component, one of *ironic.common.components*.
- `indicator` Indicator ID (as reported by *get_supported_indicators*).

Raises InvalidParameterValue if an invalid component or indicator is specified.

Raises MissingParameterValue if a required parameter is missing

Returns Current state of the indicator, one of *ironic.common.indicator_states*.

`get_mac_addresses`(*task*)
Get MAC address information for the node.

Parameters `task` A TaskManager instance containing the node to act on.

Raises UnsupportedDriverExtension

Returns A list of MAC addresses for the node

`get_secure_boot_state`(*task*)
Get the current secure boot state for the node.

NOTE: Not all drivers support this method. Older hardware may not implement that.

Parameters `task` A task from TaskManager.

Raises MissingParameterValue if a required parameter is missing

Raises DriverOperationError or its derivative in case of driver runtime error.

Raises UnsupportedDriverExtension if secure boot is not supported by the driver or the hardware

Returns Boolean

abstract get_sensors_data(*task*)
Get sensors data method.

Parameters **task** A TaskManager instance.

Raises FailedToGetSensorData when getting the sensor data fails.

Raises FailedToParseSensorData when parsing sensor data fails.

Returns

Returns a consistent format dict of sensor data grouped by sensor type, which can be processed by Ceilometer. eg,

```
{
  'Sensor Type 1': {
    'Sensor ID 1': {
      'Sensor Reading': 'current value',
      'key1': 'value1',
      'key2': 'value2'
    },
    'Sensor ID 2': {
      'Sensor Reading': 'current value',
      'key1': 'value1',
      'key2': 'value2'
    }
  },
  'Sensor Type 2': {
    'Sensor ID 3': {
      'Sensor Reading': 'current value',
      'key1': 'value1',
      'key2': 'value2'
    },
    'Sensor ID 4': {
      'Sensor Reading': 'current value',
      'key1': 'value1',
      'key2': 'value2'
    }
  }
}
```

abstract get_supported_boot_devices(*task*)
Get a list of the supported boot devices.

Parameters **task** A task from TaskManager.

Returns A list with the supported boot devices defined in *ironic.common.boot_devices*.

get_supported_boot_modes(*task*)
Get a list of the supported boot modes.

NOTE: Not all drivers support this method. Older hardware may not implement that.

Parameters **task** A task from TaskManager.

Raises UnsupportedDriverExtension if requested operation is not supported by the driver

Raises DriverOperationError or its derivative in case of driver runtime error.

Raises MissingParameterValue if a required parameter is missing

Returns A list with the supported boot modes defined in *ironic.common.boot_modes*. If boot mode support cant be determined, empty list is returned.

get_supported_indicators(*task*, *component=None*)

Get a map of the supported indicators (e.g. LEDs).

Parameters

- **task** A task from TaskManager.
- **component** If not *None*, return indicator information for just this component, otherwise return indicators for all existing components.

Returns

A dictionary of hardware components (*ironic.common.components*) as keys with values being dictionaries having indicator IDs as keys and indicator properties as values.

```
{
    'chassis': {
        'enclosure-0': {
            "readonly": true,
            "states": [
                "off",
                "on"
            ]
        }
    },
    'system':
        'blade-A': {
            "readonly": true,
            "states": [
                "pff",
                "on"
            ]
        }
    },
    'drive':
        'ssd0': {
            "readonly": true,
            "states": [
                "off",
                "on"
            ]
        }
    }
}
```

inject_nmi(*task*)
Inject NMI, Non Maskable Interrupt.

Inject NMI (Non Maskable Interrupt) for a node immediately.

Parameters **task** A TaskManager instance containing the node to act on.

Raises UnsupportedDriverExtension

interface_type = 'management'
Interface type, used for clean steps and logging.

abstract set_boot_device(*task*, *device*, *persistent=False*)
Set the boot device for a node.

Set the boot device to use on next reboot of the node.

Parameters

- **task** A task from TaskManager.
- **device** The boot device, one of *ironic.common.boot_devices*.
- **persistent** Boolean value. True if the boot device will persist to all future boots, False if not. Default: False.

Raises InvalidParameterValue if an invalid boot device is specified.

Raises MissingParameterValue if a required parameter is missing

set_boot_mode(*task*, *mode*)
Set the boot mode for a node.

Set the boot mode to use on next reboot of the node.

Drivers implementing this method are required to implement the *get_supported_boot_modes* method as well.

NOTE: Not all drivers support this method. Hardware supporting only one boot mode may not implement that.

Parameters

- **task** A task from TaskManager.
- **mode** The boot mode, one of *ironic.common.boot_modes*.

Raises InvalidParameterValue if an invalid boot mode is specified.

Raises MissingParameterValue if a required parameter is missing

Raises UnsupportedDriverExtension if requested operation is not supported by the driver

Raises DriverOperationError or its derivative in case of driver runtime error.

set_indicator_state(*task*, *component*, *indicator*, *state*)
Set indicator on the hardware component to the desired state.

Parameters

- **task** A task from TaskManager.

- **component** The hardware component, one of *ironic.common.components*.
- **indicator** Indicator ID (as reported by *get_supported_indicators*).

State Desired state of the indicator, one of *ironic.common.indicator_states*.

Raises InvalidParameterValue if an invalid component, indicator or state is specified.

Raises MissingParameterValue if a required parameter is missing

set_secure_boot_state(*task*, *state*)

Set the current secure boot state for the node.

NOTE: Not all drivers support this method. Older hardware may not implement that.

Parameters

- **task** A task from TaskManager.
- **state** A new state as a boolean.

Raises MissingParameterValue if a required parameter is missing

Raises DriverOperationError or its derivative in case of driver runtime error.

Raises UnsupportedDriverExtension if secure boot is not supported by the driver or the hardware

class ironic.drivers.base.**NetworkInterface**(**args*, ***kwargs*)

Bases: *ironic.drivers.base.BaseInterface*

Base class for network interfaces.

abstract add_cleaning_network(*task*)

Add the cleaning network to a node.

Parameters **task** A TaskManager instance.

Returns a dictionary in the form {port.uuid: neutron_port[id]}

Raises NetworkError

add_inspection_network(*task*)

Add the inspection network to the node.

Parameters **task** A TaskManager instance.

Returns a dictionary in the form {port.uuid: neutron_port[id]}

Raises NetworkError

Raises InvalidParameterValue, if the network interface configuration is invalid.

abstract add_provisioning_network(*task*)

Add the provisioning network to a node.

Parameters **task** A TaskManager instance.

Raises NetworkError

add_rescuing_network(*task*)

Add the rescuing network to the node.

Parameters **task** A TaskManager instance.

Returns a dictionary in the form {port.uuid: neutron_port[id]}

Raises NetworkError

Raises InvalidParameterValue, if the network interface configuration is invalid.

abstract configure_tenant_networks(*task*)
Configure tenant networks for a node.

Parameters **task** A TaskManager instance.

Raises NetworkError

abstract get_current_vif(*task*, *p_obj*)
Returns the currently used VIF associated with port or portgroup

We are booting the node only in one network at a time, and presence of cleaning_vif_port_id means were doing cleaning, of provisioning_vif_port_id - provisioning, of rescuing_vif_port_id - rescuing. Otherwise its a tenant network.

Parameters

- **task** A TaskManager instance.
- **p_obj** Ironic port or portgroup object.

Returns VIF ID associated with p_obj or None.

get_node_network_data(*task*)
Return network configuration for node NICs.

Gather L2 and L3 network settings from ironic port/portgroups objects and underlying network provider, then put together collected data in form of Nova network metadata (*network_data.json*) dict.

Ironic would eventually pass network configuration to the node being managed out-of-band.

Parameters **task** A TaskManager instance.

Raises InvalidParameterValue, if the network interface configuration is invalid.

Raises MissingParameterValue, if some parameters are missing.

Returns a dict holding network configuration information adhearing Nova network metadata layout (*network_data.json*).

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

interface_type = 'network'
Interface type, used for clean steps and logging.

need_power_on(*task*)
Check if ironic node must be powered on before applying network changes

Parameters **task** A TaskManager instance.

Returns Boolean.

abstract port_changed(*task*, *port_obj*)
Handle any actions required when a port changes

Parameters

- **task** A TaskManager instance.
- **port_obj** a changed Port object.

Raises Conflict, FailedToUpdateDHCPOptOnPort

abstract portgroup_changed(*task*, *portgroup_obj*)
Handle any actions required when a port changes

Parameters

- **task** A TaskManager instance.
- **portgroup_obj** a changed Port object.

Raises Conflict, FailedToUpdateDHCPOptOnPort

abstract remove_cleaning_network(*task*)
Remove the cleaning network from a node.

Parameters **task** A TaskManager instance.

Raises NetworkError

remove_inspection_network(*task*)
Removes the inspection network from a node.

Parameters **task** A TaskManager instance.

Raises NetworkError

Raises InvalidParameterValue, if the network interface configuration is invalid.

Raises MissingParameterValue, if some parameters are missing.

abstract remove_provisioning_network(*task*)
Remove the provisioning network from a node.

Parameters **task** A TaskManager instance.

remove_rescuing_network(*task*)
Removes the rescuing network from a node.

Parameters **task** A TaskManager instance.

Raises NetworkError

Raises InvalidParameterValue, if the network interface configuration is invalid.

Raises MissingParameterValue, if some parameters are missing.

abstract unconfigure_tenant_networks(*task*)
Unconfigure tenant networks for a node.

Parameters **task** A TaskManager instance.

validate(*task*)
Validates the network interface.

Parameters **task** A TaskManager instance.

Raises InvalidParameterValue, if the network interface configuration is invalid.

Raises MissingParameterValue, if some parameters are missing.

validate_inspection(*task*)
Validate that the node has required properties for inspection.

Parameters **task** A TaskManager instance with the node being checked

Raises MissingParameterValue if node is missing one or more required parameters

Raises UnsupportedDriverExtension

validate_rescue(*task*)
Validates the network interface for rescue operation.

Parameters **task** A TaskManager instance.

Raises InvalidParameterValue, if the network interface configuration is invalid.

Raises MissingParameterValue, if some parameters are missing.

abstract vif_attach(*task*, *vif_info*)
Attach a virtual network interface to a node

Parameters

- **task** A TaskManager instance.
- **vif_info** a dictionary of information about a VIF. It must have an id key, whose value is a unique identifier for that VIF.

Raises NetworkError, VifAlreadyAttached, NoFreePhysicalPorts

abstract vif_detach(*task*, *vif_id*)
Detach a virtual network interface from a node

Parameters

- **task** A TaskManager instance.
- **vif_id** A VIF ID to detach

Raises NetworkError, VifNotAttached

abstract vif_list(*task*)
List attached VIF IDs for a node

Parameters **task** A TaskManager instance.

Returns List of VIF dictionaries, each dictionary will have an id entry with the ID of the VIF.

class ironic.drivers.base.**PowerInterface**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.BaseInterface*

Interface for power-related actions.

abstract get_power_state(*task*)
Return the power state of the tasks node.

Parameters **task** A TaskManager instance containing the node to act on.

Raises MissingParameterValue if a required parameter is missing.

Returns A power state. One of *ironic.common.states*.

get_supported_power_states(*task*)
Get a list of the supported power states.

Parameters **task** A TaskManager instance containing the node to act on.

Returns A list with the supported power states defined in *ironic.common.states*.

interface_type = 'power'
Interface type, used for clean steps and logging.

abstract reboot(*task*, *timeout=None*)
Perform a hard reboot of the tasks node.

Drivers are expected to properly handle case when node is powered off by powering it on.

Parameters

- **task** A TaskManager instance containing the node to act on.
- **timeout** timeout (in seconds) positive integer (> 0) for any power state. None indicates to use default timeout.

Raises MissingParameterValue if a required parameter is missing.

abstract set_power_state(*task*, *power_state*, *timeout=None*)
Set the power state of the tasks node.

Parameters

- **task** A TaskManager instance containing the node to act on.
- **power_state** Any power state from *ironic.common.states*.
- **timeout** timeout (in seconds) positive integer (> 0) for any power state. None indicates to use default timeout.

Raises MissingParameterValue if a required parameter is missing.

supports_power_sync(*task*)
Check if power sync is supported for the given node.

If False, the conductor will simply store whatever get_power_state returns in the database instead of trying to force the expected power state.

Parameters **task** A TaskManager instance containing the node to act on.

Returns boolean, whether power sync is supported.

class ironic.drivers.base.**RAIDInterface**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.BaseInterface*

apply_configuration(*task*, *raid_config*, *create_root_volume=True*, *create_nonroot_volumes=True*, *delete_existing=True*)
Applies RAID configuration on the given node.

Parameters

- **task** A TaskManager instance.
- **raid_config** The RAID configuration to apply.

- **create_root_volume** Setting this to False indicates not to create root volume that is specified in raid_config. Default value is True.
- **create_nonroot_volumes** Setting this to False indicates not to create non-root volumes (all except the root volume) in raid_config. Default value is True.
- **delete_existing** Setting this to True indicates to delete RAID configuration prior to creating the new configuration.

Raises InvalidParameterValue, if the RAID configuration is invalid.

Returns states.DEPLOYWAIT if RAID configuration is in progress asynchronously or None if it is complete.

abstract create_configuration(*task*, *create_root_volume=True*, *create_nonroot_volumes=True*, *delete_existing=True*)

Creates RAID configuration on the given node.

This method creates a RAID configuration on the given node. It assumes that the target RAID configuration is already available in node.target_raid_config. Implementations of this interface are supposed to read the RAID configuration from node.target_raid_config. After the RAID configuration is done (either in this method OR in a call-back method), ironic.common.raid.update_raid_info() may be called to sync the nodes RAID-related information with the RAID configuration applied on the node.

Parameters

- **task** A TaskManager instance.
- **create_root_volume** Setting this to False indicates not to create root volume that is specified in the nodes target_raid_config. Default value is True.
- **create_nonroot_volumes** Setting this to False indicates not to create non-root volumes (all except the root volume) in the nodes target_raid_config. Default value is True.
- **delete_existing** Setting this to True indicates to delete RAID configuration prior to creating the new configuration.

Returns states.CLEANWAIT (cleaning) or states.DEPLOYWAIT (deployment) if RAID configuration is in progress asynchronously, or None if it is complete.

abstract delete_configuration(*task*)

Deletes RAID configuration on the given node.

This method deletes the RAID configuration on the give node. After RAID configuration is deleted, node.raid_config should be cleared by the implementation.

Parameters **task** A TaskManager instance.

Returns states.CLEANWAIT (cleaning) or states.DEPLOYWAIT (deployment) if deletion is in progress asynchronously, or None if it is complete.

get_logical_disk_properties()

Get the properties that can be specified for logical disks.

This method returns a dictionary containing the properties that can be specified for logical disks and a textual description for them.

Returns A dictionary containing properties that can be mentioned for logical disks and a textual description for them.

get_properties()
Return the properties of the interface.

Returns dictionary of <property name>:<property description> entries.

interface_type = 'raid'
Interface type, used for clean steps and logging.

validate(*task*)
Validates the RAID Interface.

This method validates the properties defined by Ironic for RAID configuration. Driver implementations of this interface can override this method for doing more validations (such as BMCs credentials).

Parameters **task** A TaskManager instance.

Raises InvalidParameterValue, if the RAID configuration is invalid.

Raises MissingParameterValue, if some parameters are missing.

validate_raid_config(*task*, *raid_config*)
Validates the given RAID configuration.

This method validates the given RAID configuration. Driver implementations of this interface can override this method to support custom parameters for RAID configuration.

Parameters

- **task** A TaskManager instance.
- **raid_config** The RAID configuration to validate.

Raises InvalidParameterValue, if the RAID configuration is invalid.

ironic.drivers.base.**RAID_APPLY_CONFIGURATION_ARGSINFO = {'create_nonroot_volumes': {'description': "Setting this to 'False' indicates not to create non-root volumes (all except the root volume) in 'raid_config'. Default value is 'True'.", 'required': False}, 'create_root_volume': {'description': "Setting this to 'False' indicates not to create root volume that is specified in 'raid_config'. Default value is 'True'.", 'required': False}, 'delete_existing': {'description': "Setting this to 'True' indicates to delete existing RAID configuration prior to creating the new configuration. Default value is 'True'.", 'required': False}, 'raid_config': {'description': 'The RAID configuration to apply.', 'required': True}}**
This may be used as the deploy_step argsinfo argument for RAID interfaces implementing an apply_configuration deploy step.

class ironic.drivers.base.**RescueInterface**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.BaseInterface*

Interface for rescue-related actions.

clean_up(*task*)
Clean up the rescue environment for the tasks node.

This is particularly useful for nodes where rescuing is asynchronous and a timeout occurs.

Parameters **task** A TaskManager instance containing the node to act on.

Returns None

interface_type = 'rescue'
Interface type, used for clean steps and logging.

abstract rescue(*task*)
Boot the tasks node into a rescue environment.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InstanceRescueFailure if node validation or rescue operation fails.

Returns states.RESCUEWAIT if rescue is in progress asynchronously or states.RESCUE if it is complete.

abstract unrescue(*task*)
Tear down the rescue environment, and return to normal.

Parameters **task** A TaskManager instance containing the node to act on.

Raises InstanceUnrescueFailure if node validation or unrescue operation fails.

Returns states.ACTIVE if it is successful.

class ironic.drivers.base.**StorageInterface**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.BaseInterface*

Base class for storage interfaces.

abstract attach_volumes(*task*)
Informs the storage subsystem to attach all volumes for the node.

Parameters **task** A TaskManager instance.

Raises UnsupportedDriverExtension

abstract detach_volumes(*task*)
Informs the storage subsystem to detach all volumes for the node.

Parameters **task** A TaskManager instance.

Raises UnsupportedDriverExtension

interface_type = 'storage'
Interface type, used for clean steps and logging.

abstract should_write_image(*task*)
Determines if deploy should perform the image write-out.

Parameters **task** A TaskManager instance.

Returns Boolean value to indicate if the interface expects the image to be written by Ironic.

Raises UnsupportedDriverExtension

class ironic.drivers.base.**VendorInterface**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.BaseInterface*

Interface for all vendor passthru functionality.

Additional vendor- or driver-specific capabilities should be implemented as a method in the class inheriting from this class and use the @passthru or @driver_passthru decorators.

Methods decorated with @driver_passthru should be short-lived because it is a blocking call.

driver_validate(*method*, ***kwargs*)
Validate driver-vendor-passthru actions.

If invalid, raises an exception; otherwise returns None.

Parameters

- **method** method to be validated
- **kwargs** info for action.

Raises MissingParameterValue if kwargs does not contain certain parameter.

Raises InvalidParameterValue if parameter does not match.

interface_type = 'vendor'
Interface type, used for clean steps and logging.

abstract validate(*task*, *method=None*, ***kwargs*)
Validate vendor-specific actions.

If invalid, raises an exception; otherwise returns None.

Parameters

- **task** A task from TaskManager.
- **method** Method to be validated
- **kwargs** Info for action.

Raises UnsupportedDriverExtension if method can not be mapped to the supported interfaces.

Raises InvalidParameterValue if kwargs does not contain method.

Raises MissingParameterValue

class ironic.drivers.base.**VendorMetadata**(*method*, *metadata*)
Bases: tuple

metadata
Alias for field number 1

method
Alias for field number 0

ironic.drivers.base.**cache_bios_settings**(*func*)
A decorator to cache bios settings after running the function.

Parameters **func** Function or method to wrap.

ironic.drivers.base.**clean_step**(*priority*, *abortable=False*, *argsinfo=None*, *requires_ramdisk=True*)
Decorator for cleaning steps.

Cleaning steps may be used in manual or automated cleaning.

For automated cleaning, only steps with priorities greater than 0 are used. These steps are ordered by priority from highest value to lowest value. For steps with the same priority, they are ordered by driver interface priority (see conductor.steps.CLEANING_INTERFACE_PRIORITY). execute_clean_step() will be called on each step.

For manual cleaning, the clean steps will be executed in a similar fashion to automated cleaning, but the steps and order of execution must be explicitly specified by the user when invoking the cleaning API.

Decorated clean steps must take as the only positional argument, a TaskManager object. Clean steps used in manual cleaning may also take keyword variable arguments (as described in argsinfo).

Clean steps can be either synchronous or asynchronous. If the step is synchronous, it should return *None* when finished, and the conductor will continue on to the next step. While the clean step is executing, the node will be in *states.CLEANING* provision state. If the step is asynchronous, the step should return *states.CLEANWAIT* to the conductor before it starts the asynchronous work. When the step is complete, the step should make an RPC call to *continue_node_clean* to move to the next step in cleaning. The node will be in *states.CLEANWAIT* provision state during the asynchronous work.

Examples:

```
class MyInterface(base.BaseInterface):
    # CONF.example_cleaning_priority should be an int CONF option
    @base.clean_step(priority=CONF.example_cleaning_priority)
    def example_cleaning(self, task):
        # do some cleaning

    @base.clean_step(priority=0, abortable=True, argsinfo=
                     {'size': {'description': 'size of widget (MB)',
                               'required': True}})
    def advanced_clean(self, task, **kwargs):
        # do some advanced cleaning
```

Parameters

- **priority** an integer priority, should be a CONF option
- **abortable** Boolean value. Whether the clean step is abortable or not; defaults to False.
- **argsinfo** a dictionary of keyword arguments where key is the name of the argument and value is a dictionary as follows:

```
'description': <description>. Required. This should␣
↪include
               possible values.
'required': Boolean. Optional; default is False. True if␣
↪this
            argument is required.  If so, it must be␣
↪specified in
            the clean request; false if it is optional.
```

- **requires_ramdisk** Whether this step requires the ramdisk to be running. Should be set to False for purely out-of-band steps.

Raises *InvalidParameterValue* if any of the arguments are invalid

ironic.drivers.base.**deploy_step**(*priority*, *argsinfo=None*)

Decorator for deployment steps.

Only steps with priorities greater than 0 are used. These steps are ordered by priority from highest value to lowest value. For steps with the same priority, they are ordered by driver interface priority (see conductor.steps.DEPLOYING_INTERFACE_PRIORITY). execute_deploy_step() will be called on each step.

Decorated deploy steps must take as the only positional argument, a TaskManager object.

Deploy steps can be either synchronous or asynchronous. If the step is synchronous, it should return *None* when finished, and the conductor will continue on to the next step. While the deploy step is executing, the node will be in *states.DEPLOYING* provision state. If the step is asynchronous, the step should return *states.DEPLOYWAIT* to the conductor before it starts the asynchronous work. When the step is complete, the step should make an RPC call to *continue_node_deploy* to move to the next step in deployment. The node will be in *states.DEPLOYWAIT* provision state during the asynchronous work.

Examples:

```
class MyInterface(base.BaseInterface):
    @base.deploy_step(priority=100)
    def example_deploying(self, task):
        # do some deploying
```

Parameters

- **priority** an integer (>=0) priority; used for determining the order in which the step is run in the deployment process.
- **argsinfo** a dictionary of keyword arguments where key is the name of the argument and value is a dictionary as follows:

```
'description': <description>. Required. This should␣
↪include
               possible values.
'required': Boolean. Optional; default is False. True if␣
↪this
            argument is required.  If so, it must be␣
↪specified in
            the deployment request; false if it is␣
↪optional.
```

Raises *InvalidParameterValue* if any of the arguments are invalid

ironic.drivers.base.**driver_passthru**(*http_methods*, *method=None*, *async_call=True*, *description=None*, *attach=False*)

ironic.drivers.base.**passthru**(*http_methods*, *method=None*, *async_call=True*, *description=None*, *attach=False*, *require_exclusive_lock=True*)

ironic.drivers.base.**verify_step**(*priority*)

Decorator for verify steps.

Only steps with priorities greater than 0 are used. These steps are ordered by priority from highest value to lowest value. For steps with the same priority, they are ordered by driver interface priority (see conductor.steps.VERIFY_INTERFACE_PRIORITY). execute_verify_step() will be called on each step.

Decorated verify steps must take as the only positional argument, a TaskManager object.

Verify steps are synchronous and should return *None* when finished, and the conductor will continue on to the next step. While the verify step is executing, the node will be in *states.VERIFYING* provision state.

Examples:

```
class MyInterface(base.BaseInterface):
    @base.verify_step(priority=100)
    def example_verifying(self, task):
        # do some verifying
```

Parameters **priority** an integer (>=0) priority; used for determining the order in which the step is run in the verification process.

Raises *InvalidParameterValue* if any of the arguments are invalid

ironic.drivers.drac module

DRAC Driver for remote system management using Dell Remote Access Card.

class ironic.drivers.drac.**IDRACHardware**

Bases: *ironic.drivers.generic.GenericHardware*

integrated Dell Remote Access Controller hardware type

property supported_bios_interfaces

List of supported bios interfaces.

property supported_boot_interfaces

List of supported boot interfaces.

property supported_inspect_interfaces

List of supported inspect interfaces.

property supported_management_interfaces

List of supported management interfaces.

property supported_power_interfaces

List of supported power interfaces.

property supported_raid_interfaces

List of supported raid interfaces.

property supported_vendor_interfaces

List of supported vendor interfaces.

ironic.drivers.fake_hardware module

Fake hardware type.

class ironic.drivers.fake_hardware.**FakeHardware**
Bases: *ironic.drivers.generic.GenericHardware*

Fake hardware type.

This hardware type is special-cased in the driver factory to bypass compatibility verification. Thus, supported_* methods here are only for calculating the defaults, not for actual check.

All fake implementations are still expected to be enabled in the configuration.

property supported_bios_interfaces
List of classes of supported bios interfaces.

property supported_boot_interfaces
List of classes of supported boot interfaces.

property supported_console_interfaces
List of classes of supported console interfaces.

property supported_deploy_interfaces
List of classes of supported deploy interfaces.

property supported_inspect_interfaces
List of classes of supported inspect interfaces.

property supported_management_interfaces
List of classes of supported management interfaces.

property supported_power_interfaces
List of classes of supported power interfaces.

property supported_raid_interfaces
List of classes of supported raid interfaces.

property supported_rescue_interfaces
List of classes of supported rescue interfaces.

property supported_storage_interfaces
List of classes of supported storage interfaces.

property supported_vendor_interfaces
List of classes of supported rescue interfaces.

ironic.drivers.generic module

Generic hardware types.

class ironic.drivers.generic.**GenericHardware**
Bases: *ironic.drivers.hardware_type.AbstractHardwareType*

Abstract base class representing generic hardware.

This class provides reasonable defaults for all of the interfaces.

property supported_boot_interfaces
List of supported boot interfaces.

property supported_deploy_interfaces
List of supported deploy interfaces.

property supported_inspect_interfaces
List of supported inspect interfaces.

property supported_network_interfaces
List of supported network interfaces.

property supported_raid_interfaces
List of supported raid interfaces.

property supported_rescue_interfaces
List of supported rescue interfaces.

property supported_storage_interfaces
List of supported storage interfaces.

class ironic.drivers.generic.**ManualManagementHardware**
Bases: *ironic.drivers.generic.GenericHardware*

Hardware type that uses manual power and boot management.

Using this hardware type assumes that an operator manages reboot and setting boot devices manually. This hardware type should only be used when no suitable hardware type exists in ironic, or the existing hardware type misbehaves for any reason.

property supported_management_interfaces
List of supported management interfaces.

property supported_power_interfaces
List of supported power interfaces.

property supported_vendor_interfaces
List of supported vendor interfaces.

ironic.drivers.hardware_type module

Abstract base class for all hardware types.

class ironic.drivers.hardware_type.**AbstractHardwareType**
Bases: object

Abstract base class for all hardware types.

Hardware type is a family of hardware supporting the same set of interfaces from the ironic standpoint. This can be as wide as all hardware supporting the IPMI protocol or as narrow as several hardware models supporting some specific interfaces.

A hardware type defines an ordered list of supported implementations for each driver interface (power, deploy, etc).

get_properties()
Get the properties of the hardware type.

Note that this returns properties for the default interface of each type, for this hardware type. Since this is not node-aware, interface overrides cant be detected.

Returns dictionary of <property name>:<property description> entries.

supported = True
Whether hardware is supported by the community.

property supported_bios_interfaces
List of supported bios interfaces.

abstract property supported_boot_interfaces
List of supported boot interfaces.

property supported_console_interfaces
List of supported console interfaces.

abstract property supported_deploy_interfaces
List of supported deploy interfaces.

property supported_inspect_interfaces
List of supported inspect interfaces.

abstract property supported_management_interfaces
List of supported management interfaces.

property supported_network_interfaces
List of supported network interfaces.

abstract property supported_power_interfaces
List of supported power interfaces.

property supported_raid_interfaces
List of supported raid interfaces.

property supported_rescue_interfaces
List of supported rescue interfaces.

property supported_storage_interfaces
List of supported storage interfaces.

property supported_vendor_interfaces
List of supported vendor interfaces.

ironic.drivers.ibmc module

iBMC Driver for managing HUAWEI V5 series rack servers such as 2288H V5, CH121 V5.

class ironic.drivers.ibmc.**IBMCHardware**
Bases: *ironic.drivers.generic.GenericHardware*

Huawei iBMC hardware type.

property supported_management_interfaces
List of supported management interfaces.

property supported_power_interfaces
List of supported power interfaces.

property supported_raid_interfaces
List of supported raid interfaces.

property supported_vendor_interfaces
List of supported vendor interfaces.

ironic.drivers.ilo module

iLO Driver for managing HP Proliant Gen8 and above servers.

class ironic.drivers.ilo.**Ilo5Hardware**
Bases: *ironic.drivers.ilo.IloHardware*

iLO5 hardware type.

iLO5 hardware type is targeted for iLO5 based Proliant Gen10 servers.

property supported_boot_interfaces
List of supported boot interfaces.

property supported_management_interfaces
List of supported management interfaces.

property supported_raid_interfaces
List of supported raid interfaces.

class ironic.drivers.ilo.**IloHardware**
Bases: *ironic.drivers.generic.GenericHardware*

iLO hardware type.

iLO hardware type is targeted for iLO 4 based Proliant Gen8 and Gen9 servers.

property supported_bios_interfaces
List of supported bios interfaces.

property supported_boot_interfaces
List of supported boot interfaces.

property supported_console_interfaces
List of supported console interfaces.

property supported_inspect_interfaces
List of supported inspect interfaces.

property supported_management_interfaces
List of supported management interfaces.

property supported_power_interfaces
List of supported power interfaces.

property supported_vendor_interfaces
List of supported power interfaces.

ironic.drivers.intel_ipmi module

class ironic.drivers.intel_ipmi.**IntelIPMIHardware**
Bases: *ironic.drivers.ipmi.IPMIHardware*

Intel IPMI hardware type.

Uses `ipmitool` to implement power and management. Provides serial console implementations via `shellinabox` or `socat`. Supports Intel SST-PP feature.

property supported_management_interfaces
List of supported management interfaces.

ironic.drivers.ipmi module

Hardware type for IPMI (using ipmitool).

class ironic.drivers.ipmi.**IPMIHardware**
Bases: *ironic.drivers.generic.GenericHardware*

IPMI hardware type.

Uses `ipmitool` to implement power and management. Provides serial console implementations via `shellinabox` or `socat`.

property supported_console_interfaces
List of supported console interfaces.

property supported_management_interfaces
List of supported management interfaces.

property supported_power_interfaces
List of supported power interfaces.

property supported_vendor_interfaces
List of supported vendor interfaces.

ironic.drivers.irmc module

iRMC Driver for managing FUJITSU PRIMERGY BX S4 or RX S8 generation of FUJITSU PRIMERGY servers, and above servers.

class ironic.drivers.irmc.**IRMCHardware**
Bases: *ironic.drivers.generic.GenericHardware*

iRMC hardware type.

iRMC hardware type is targeted for FUJITSU PRIMERGY servers which have iRMC S4 management system.

property supported_bios_interfaces
List of supported bios interfaces.

property supported_boot_interfaces
List of supported boot interfaces.

property supported_console_interfaces
List of supported console interfaces.

property supported_inspect_interfaces
List of supported inspect interfaces.

property supported_management_interfaces
List of supported management interfaces.

property supported_power_interfaces
List of supported power interfaces.

property supported_raid_interfaces
List of supported raid interfaces.

ironic.drivers.redfish module

class ironic.drivers.redfish.**RedfishHardware**
Bases: *ironic.drivers.generic.GenericHardware*

Redfish hardware type.

property supported_bios_interfaces
List of supported bios interfaces.

property supported_boot_interfaces
List of supported boot interfaces.

property supported_inspect_interfaces
List of supported power interfaces.

property supported_management_interfaces
List of supported management interfaces.

property supported_power_interfaces
List of supported power interfaces.

property supported_raid_interfaces
List of supported raid interfaces.

property supported_vendor_interfaces
List of supported vendor interfaces.

ironic.drivers.snmp module

SNMP hardware types.

class ironic.drivers.snmp.**SNMPHardware**
Bases: *ironic.drivers.generic.GenericHardware*

SNMP Hardware type

property supported_management_interfaces
List of supported management interfaces.

property supported_power_interfaces
List of supported power interfaces.

ironic.drivers.utils module

class ironic.drivers.utils.**MixinVendorInterface**(**args*, ***kwargs*)
Bases: *ironic.drivers.base.VendorInterface*

Wrapper around multiple VendorInterfaces.

get_properties()
Return the properties from all the VendorInterfaces.

Returns a dictionary of <property_name>:<property_description> entries.

validate(*task*, *method*, ***kwargs*)
Call validate on the appropriate interface only.

Raises UnsupportedDriverExtension if method can not be mapped to the supported interfaces.

Raises InvalidParameterValue if method is invalid.

Raises MissingParameterValue if missing method or parameters in kwargs.

ironic.drivers.utils.**add_node_capability**(*task*, *capability*, *value*)
Add capability to nodes capabilities property.

If capability is already present, then a duplicate entry will be added.

Parameters

- **task** Task object.
- **capability** Capability key.
- **value** Capability value.

ironic.drivers.utils.**capabilities_to_dict**(*capabilities*)
Parse the capabilities string into a dictionary

Parameters **capabilities** the capabilities of the node as a formatted string.

Raises InvalidParameterValue if capabilities is not an string or has a malformed value

ironic.drivers.utils.**collect_ramdisk_logs**(*node*, *label=None*)
Collect and store the system logs from the IPA ramdisk.

Collect and store the system logs from the IPA ramdisk. This method makes a call to the IPA ramdisk to collect the logs and store it according to the configured storage backend.

Parameters

- **node** A node object.
- **label** A string to label the log file such as a clean step name.

ironic.drivers.utils.**ensure_next_boot_device**(*task*, *driver_info*)
Ensure boot from correct device if persistent is True

If ipmi_force_boot_device is True and is_next_boot_persistent, set to boot from correct device, else unset is_next_boot_persistent field.

Parameters

- **task** Node object.
- **driver_info** Node driver_info.

ironic.drivers.utils.**force_persistent_boot**(*task*, *device*, *persistent*)
Set persistent boot device to driver_internal_info

If persistent is True set persistent_boot_device field to the boot device and reset persistent to False, else set is_next_boot_persistent to False.

Parameters

- **task** Task object.
- **device** Boot device.
- **persistent** Whether next boot is persistent or not.

ironic.drivers.utils.**get_agent_iso**(*node*, *mode='deploy'*, *deprecated_prefix=None*)
Get the agent ISO image.

ironic.drivers.utils.**get_agent_kernel_ramdisk**(*node*, *mode='deploy'*, *deprecated_prefix=None*)
Get the agent kernel/ramdisk as a dictionary.

ironic.drivers.utils.**get_field**(*node*, *name*, *deprecated_prefix=None*, *use_conf=False*, *collection='driver_info'*)
Get a driver_info field with deprecated prefix.

ironic.drivers.utils.**get_kernel_append_params**(*node*, *default*)
Get the applicable kernel params.

The locations are checked in this order:

1. The nodes instance_info.
2. The nodes driver_info.
3. Configuration.

Parameters

- **node** Node object.
- **default** Default value.

ironic.drivers.utils.**get_node_capability**(*node*, *capability*)
Returns capability value from nodes capabilities property.

Parameters

- **node** Node object.
- **capability** Capability key.

Returns Capability value. If capability is not present, then return None

ironic.drivers.utils.**get_node_mac_addresses**(*task*)
Get all MAC addresses for the ports belonging to this tasks node.

Parameters **task** a TaskManager instance containing the node to act on.

Returns A list of MAC addresses in the format xx:xx:xx:xx:xx:xx.

ironic.drivers.utils.**get_ramdisk_logs_file_name**(*node*, *label=None*)
Construct the log file name.

Parameters

- **node** A node object.
- **label** A string to label the log file such as a clean step name.

Returns The log file name.

ironic.drivers.utils.**need_prepare_ramdisk**(*node*)
Check if node needs preparing ramdisk

Parameters **node** Node to check for

Returns True if need to prepare ramdisk, otherwise False

ironic.drivers.utils.**normalize_mac**(*mac*)
Remove - and : characters and lowercase the MAC string.

Parameters **mac** MAC address to normalize.

Returns Normalized MAC address string.

ironic.drivers.utils.**store_ramdisk_logs**(*node*, *logs*, *label=None*)
Store the ramdisk logs.

This method stores the ramdisk logs according to the configured storage backend.

Parameters

- **node** A node object.
- **logs** A gzipped and base64 encoded string containing the logs archive.
- **label** A string to label the log file such as a clean step name.

Raises OSError if the directory to save the logs cannot be created.

Raises IOError when the logs cant be saved to the local file system.

Raises SwiftOperationError, if any operation with Swift fails.

ironic.drivers.xclarity module

XClarity Driver and supporting meta-classes.

class ironic.drivers.xclarity.**XClarityHardware**
Bases: *ironic.drivers.generic.GenericHardware*

XClarity hardware type.

property supported_management_interfaces
List of supported management interfaces.

property supported_power_interfaces
List of supported power interfaces.

Module contents

ironic.objects package

Submodules

ironic.objects.allocation module

class ironic.objects.allocation.**Allocation**(*context=None*, ***kwargs*)
Bases: *ironic.objects.base.IronicObject*, oslo_versionedobjects.base.VersionedObjectDictCompat

VERSION = '1.1'

property candidate_nodes

property conductor_affinity

create(*context=None*)
Create a Allocation record in the DB.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Allocation(context)

Raises AllocationDuplicateName, AllocationAlreadyExists

property created_at

dbapi = <oslo_db.api.DBAPI object>

destroy(*context=None*)
Delete the Allocation from the DB.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Allocation(context)

Raises AllocationNotFound

property extra

fields = {'candidate_nodes': List(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'conductor_affinity': Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'created_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'extra': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'id':
Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False),
'last_error': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'name':
String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'node_id': Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'owner': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'resource_class': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'state': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'traits': List(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'updated_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid':
UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

classmethod get(*context*, *allocation_ident*)
Find an allocation by its ID, UUID or name.

Parameters

- **allocation_ident** The ID, UUID or name of an allocation.
- **context** Security context

Returns An *Allocation* object.

Raises InvalidIdentity

classmethod get_by_id(*context*, *allocation_id*)
Find an allocation by its integer ID.

Parameters

- **cls** the *Allocation*
- **context** Security context
- **allocation_id** The ID of an allocation.

Returns An *Allocation* object.

Raises AllocationNotFound

classmethod get_by_name(*context*, *name*)
Find an allocation based by its name.

Parameters

- **cls** the *Allocation*
- **context** Security context
- **name** The name of an allocation.

Returns An *Allocation* object.

Raises AllocationNotFound

classmethod get_by_uuid(*context*, *uuid*)
Find an allocation by its UUID.

Parameters

- **cls** the *Allocation*
- **context** Security context
- **uuid** The UUID of an allocation.

Returns An *Allocation* object.

Raises AllocationNotFound

property id

property last_error

classmethod list(*context*, *filters=None*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)
Return a list of Allocation objects.

Parameters

- **cls** the *Allocation*
- **context** Security context.
- **filters** Filters to apply.
- **limit** Maximum number of resources to return in a single result.
- **marker** Pagination marker for large data sets.
- **sort_key** Column to sort results by.
- **sort_dir** Direction to sort. asc or desc.

Returns A list of *Allocation* object.

Raises InvalidParameterValue

property name

property node_id

property owner

refresh(*context=None*)

Loads updates for this Allocation.

Loads an allocation with the same uuid from the database and checks for updated attributes. Updates are applied from the loaded allocation column by column, if there are any updates.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Allocation(context)

Raises AllocationNotFound

property resource_class

save(*context=None*)

Save updates to this Allocation.

Updates will be made column by column based on the result of self.what_changed().

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Allocation(context)

Raises AllocationNotFound, AllocationDuplicateName

property state

property traits

property updated_at

property uuid

class ironic.objects.allocation.**AllocationCRUDNotification**(*context=None*, ***kwargs*)

Bases: *ironic.objects.notification.NotificationBase*

Notification when ironic creates, updates or deletes an allocation.

VERSION = '1.0'

property created_at

property event_type

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'event_type': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'level': NotificationLevel(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('debug', 'info', 'warning', 'error', 'critical')), 'payload': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'publisher': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property level

property payload

property publisher

property updated_at

class ironic.objects.allocation.**AllocationCRUDPayload**(*allocation*, *node_uuid=None*)

Bases: *ironic.objects.notification.NotificationPayloadBase*

SCHEMA = {'candidate_nodes': ('allocation', 'candidate_nodes'), 'created_at': ('allocation', 'created_at'), 'extra': ('allocation', 'extra'), 'last_error': ('allocation', 'last_error'), 'name': ('allocation', 'name'), 'owner': ('allocation', 'owner'), 'resource_class': ('allocation', 'resource_class'), 'state': ('allocation', 'state'), 'traits': ('allocation', 'traits'), 'updated_at': ('allocation', 'updated_at'), 'uuid': ('allocation', 'uuid')}

VERSION = '1.1'

property candidate_nodes

property created_at

property extra

fields = {'candidate_nodes': List(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'created_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'extra': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'last_error': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'name':
String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'node_uuid': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'owner': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'resource_class': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'state': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'traits': List(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'updated_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid':
UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property last_error

property name

property node_uuid

property owner

property resource_class

property state

property traits

property updated_at

property uuid

ironic.objects.base module

Ironic common internal object model

class ironic.objects.base.**IronicObject**(*context=None*, ***kwargs*)

Bases: oslo_versionedobjects.base.VersionedObject

Base class and object factory.

This forms the base of all objects that can be remoted or instantiated via RPC. Simply defining a class that inherits from this base class will make it remotely instantiatable. Objects should implement the necessary get classmethod routines as well as save object methods as appropriate.

OBJ_PROJECT_NAMESPACE = 'ironic'

OBJ_SERIAL_NAMESPACE = 'ironic_object'

as_dict()
Return the object represented as a dict.

The returned object is JSON-serialisable.

convert_to_version(*target_version*, *remove_unavailable_fields=True*)
Convert this object to the target version.

Convert the object to the target version. The target version may be the same, older, or newer than the version of the object. This is used for DB interactions as well as for serialization/deserialization.

The remove_unavailable_fields flag is used to distinguish these two cases:

1) For serialization/deserialization, we need to remove the unavailable fields, because the service receiving the object may not know about these fields. remove_unavailable_fields is set to True in this case.

2) For DB interactions, we need to set the unavailable fields to their appropriate values so that these fields are saved in the DB. (If they are not set, the VersionedObject magic will not know to save/update them to the DB.) remove_unavailable_fields is set to False in this case.

_convert_to_version() does the actual work.

Parameters

- **target_version** the desired version of the object
- **remove_unavailable_fields** True to remove fields that are unavailable in the target version; set this to True when (de)serializing. False to set the unavailable fields to appropriate values; set this to False for DB interactions.

do_version_changes_for_db()
Change the object to the version needed for the database.

If needed, this changes the object (modifies object fields) to be in the correct version for saving to the database.

The version used to save the object in the DB is determined as follows:

- If the object is pinned, we save the object in the pinned version. Since it is pinned, we must not save in a newer version, in case a rolling upgrade is happening and some services are still using the older version of ironic, with no knowledge of this newer version.
- If the object isnt pinned, we save the object in the latest version.

Because the object may be converted to a different object version, this method must only be called just before saving the object to the DB.

Returns a dictionary of changed fields and their new values (could be an empty dictionary). These are the fields/values of the object that would be saved to the DB.

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

classmethod get_target_version()
Returns the target version for this object.

This is the version in which the object should be manipulated, e.g. sent over the wire via RPC or saved in the DB.

Returns if pinned, returns the version of this object corresponding to the pin. Otherwise, returns the version of the object.

Raises ovo_exception.IncompatibleObjectVersion

obj_refresh(*loaded_object*)
Applies updates for objects that inherit from base.IronicObject.

Checks for updated attributes in an object. Updates are applied from the loaded object column by column in comparison with the current object.

classmethod supports_version(*version*)
Return whether this object supports a particular version.

Check the requested version against the objects target version. The target version may not be the latest version during an upgrade, when object versions are pinned.

Parameters **version** A tuple representing the version to check

Returns Whether the version is supported

Raises ovo_exception.IncompatibleObjectVersion

class ironic.objects.base.**IronicObjectListBase**(**args*, ***kwargs*)
Bases: oslo_versionedobjects.base.ObjectListBase

as_dict()
Return the object represented as a dict.

The returned object is JSON-serialisable.

class ironic.objects.base.**IronicObjectRegistry**(**args*, ***kwargs*)
Bases: oslo_versionedobjects.base.VersionedObjectRegistry

registration_hook(*cls*, *index*)

class ironic.objects.base.**IronicObjectSerializer**(*is_server=False*)
Bases: oslo_versionedobjects.base.VersionedObjectSerializer

OBJ_BASE_CLASS
alias of *ironic.objects.base.IronicObject*

serialize_entity(*context*, *entity*)
Serialize the entity.

This serializes the entity so that it can be sent over e.g. RPC. A serialized entity for an IronicObject is a dictionary with keys: ironic_object.namespace, ironic_object.data, ironic_object.name, ironic_object.version, and ironic_object.changes.

We assume that the client (ironic-API) is always talking to a server (ironic-conductor) that is running the same or a newer release than the client. The client doesnt need to downgrade

any IronicObjects when sending them over RPC. The server, on the other hand, will need to do so if the server is pinned and the target version of an IronicObject is older than the latest version of that Object.

(Internally, the services deal with the latest versions of objects so we know that these objects are always in the latest versions.)

Parameters

- **context** security context
- **entity** the entity to be serialized; may be an IronicObject

Returns the serialized entity

Raises ovo_exception.IncompatibleObjectVersion (via .get_target_version())

ironic.objects.base.**max_version**(*versions*)

Return the maximum version in the list.

Parameters **versions** a list of (string) versions; assumed to have at least one entry

Returns the maximum version (string)

ironic.objects.bios module

class ironic.objects.bios.**BIOSSetting**(*context=None*, ***kwargs*)

Bases: *ironic.objects.base.IronicObject*

VERSION = '1.1'

property allowable_values

property attribute_type

create(*context=None*)

Create a BIOS Setting record in DB.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: BIOSSetting(context)

Raises NodeNotFound if the node id is not found.

Raises BIOSSettingAlreadyExists if the setting record already exists.

property created_at

dbapi = <oslo_db.api.DBAPI object>

classmethod delete(*context*, *node_id*, *name*)

Delete a BIOS Setting based on its node_id and name.

Parameters

- **context** Security context.
- **node_id** The node id.
- **name** BIOS setting name to be deleted.

Raises NodeNotFound if the node id is not found.

Raises BIOSSettingNotFound if the bios setting name is not found.

fields = {'allowable_values': List(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'attribute_type': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'lower_bound': Integer(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'max_length': Integer(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'min_length': Integer(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'name': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'node_id': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'read_only': Boolean(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'reset_required': Boolean(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'unique': Boolean(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'upper_bound': Integer(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'value': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

classmethod get(*context*, *node_id*, *name*)

Get a BIOS Setting based on its node_id and name.

Parameters

- **context** Security context.
- **node_id** The node id.
- **name** BIOS setting name to be retrieved.

Raises NodeNotFound if the node id is not found.

Raises BIOSSettingNotFound if the bios setting name is not found.

Returns A :class:BIOSSetting object.

property lower_bound

property max_length

property min_length

property name

property node_id

property read_only

registry_fields = ('attribute_type', 'allowable_values', 'lower_bound', 'max_length', 'min_length', 'read_only', 'reset_required', 'unique', 'upper_bound')

property reset_required

save(*context=None*)
Save BIOS Setting update in DB.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: BIOSSetting(context)

Raises NodeNotFound if the node id is not found.

Raises BIOSSettingNotFound if the bios setting name is not found.

property unique

property updated_at

property upper_bound

property value

class ironic.objects.bios.**BIOSSettingList**(**args*, ***kwargs*)
Bases: *ironic.objects.base.IronicObjectListBase*, *ironic.objects.base.IronicObject*

VERSION = '1.0'

classmethod create(*context*, *node_id*, *settings*)
Create a list of BIOS Setting records in DB.

Parameters

- **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: BIOSSetting(context)
- **node_id** The node id.
- **settings** A list of bios settings.

Raises NodeNotFound if the node id is not found.

Raises BIOSSettingAlreadyExists if any of the setting records already exists.

Returns A list of BIOSSetting objects.

property created_at

dbapi = <oslo_db.api.DBAPI object>

classmethod delete(*context*, *node_id*, *names*)
Delete BIOS Settings based on node_id and names.

Parameters

- **context** Security context.

- **node_id** The node id.
- **names** List of BIOS setting names to be deleted.

Raises NodeNotFound if the node id is not found.

Raises BIOSSettingNotFound if any of BIOS setting fails to delete.

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'objects': List(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

classmethod get_by_node_id(*context*, *node_id*)

Get BIOS Setting based on node_id.

Parameters

- **context** Security context.
- **node_id** The node id.

Raises NodeNotFound if the node id is not found.

Returns A list of BIOSSetting objects.

property objects

classmethod save(*context*, *node_id*, *settings*)

Save a list of BIOS Setting updates in DB.

Parameters

- **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: BIOSSetting(context)
- **node_id** The node id.
- **settings** A list of bios settings.

Raises NodeNotFound if the node id is not found.

Raises BIOSSettingNotFound if any of the bios setting names is not found.

Returns A list of BIOSSetting objects.

classmethod sync_node_setting(*context*, *node_id*, *settings*)

Returns lists of create/update/delete/unchanged settings.

This method sync with bios_settings database table and sorts out four lists of create/update/delete/unchanged settings.

Parameters

- **context** Security context.
- **node_id** The node id.
- **settings** BIOS settings to be synced.

Returns A 4-tuple of lists of BIOS settings to be created, updated, deleted and unchanged.

property updated_at

ironic.objects.chassis module

class ironic.objects.chassis.**Chassis**(*context=None*, ***kwargs*)

Bases: *ironic.objects.base.IronicObject*, oslo_versionedobjects.base.VersionedObjectDictCompat

VERSION = '1.3'

create(*context=None*)

Create a Chassis record in the DB.

Column-wise updates will be made based on the result of self.what_changed(). If target_power_state is provided, it will be checked against the in-database copy of the chassis before updates are made.

Parameters context Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Chassis(context)

property created_at

dbapi = <oslo_db.api.DBAPI object>

property description

destroy(*context=None*)

Delete the Chassis from the DB.

Parameters context Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Chassis(context)

property extra

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'description': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'extra': FlexibleDict(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'id': Integer(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid': UUID(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

classmethod get(*context*, *chassis_id*)

Find a chassis based on its id or uuid and return a Chassis object.

Parameters

- **context** Security context
- **chassis_id** the id *or* uuid of a chassis.

Returns a *Chassis* object.

classmethod get_by_id(*context*, *chassis_id*)

Find a chassis based on its integer ID and return a Chassis object.

Parameters

- **cls** the *Chassis*
- **context** Security context
- **chassis_id** the ID of a chassis.

Returns a *Chassis* object.

classmethod get_by_uuid(*context*, *uuid*)

Find a chassis based on UUID and return a *Chassis* object.

Parameters

- **cls** the *Chassis*
- **context** Security context
- **uuid** the UUID of a chassis.

Returns a *Chassis* object.

property id

classmethod list(*context*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)

Return a list of Chassis objects.

Parameters

- **cls** the *Chassis*
- **context** Security context.
- **limit** maximum number of resources to return in a single result.
- **marker** pagination marker for large data sets.
- **sort_key** column to sort results by.
- **sort_dir** direction to sort. asc or desc.

Returns a list of *Chassis* object.

refresh(*context=None*)

Loads and applies updates for this Chassis.

Loads a *Chassis* with the same uuid from the database and checks for updated attributes. Updates are applied from the loaded chassis column by column, if there are any updates.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Chassis(context)

save(*context=None*)
Save updates to this Chassis.

Updates will be made column by column based on the result of self.what_changed().

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Chassis(context)

property updated_at

property uuid

class ironic.objects.chassis.ChassisCRUDNotification(*context=None*, ***kwargs*)
Bases: *ironic.objects.notification.NotificationBase*

Notification emitted when ironic creates, updates, deletes a chassis.

VERSION = '1.0'

property created_at

property event_type

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'event_type': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'level': NotificationLevel(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('debug', 'info', 'warning', 'error', 'critical')), 'payload': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'publisher': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property level

property payload

property publisher

property updated_at

class ironic.objects.chassis.ChassisCRUDPayload(*chassis*, ***kwargs*)
Bases: *ironic.objects.notification.NotificationPayloadBase*

SCHEMA = {'created_at': ('chassis', 'created_at'), 'description': ('chassis', 'description'), 'extra': ('chassis', 'extra'), 'updated_at': ('chassis', 'updated_at'), 'uuid': ('chassis', 'uuid')}

VERSION = '1.0'

property created_at

property description

property extra

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'description': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'extra': FlexibleDict(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid': UUID(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False)}

property updated_at

property uuid

ironic.objects.conductor module

class ironic.objects.conductor.**Conductor**(*context=None*, ***kwargs*)

Bases: *ironic.objects.base.IronicObject*, oslo_versionedobjects.base.VersionedObjectDictCompat

VERSION = '1.3'

property conductor_group

property created_at

dbapi = <oslo_db.api.DBAPI object>

property drivers

fields = {'conductor_group': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'drivers': List(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'hostname': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'id': Integer(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

classmethod **get_by_hostname**(*context*, *hostname*, *online=True*)

Get a Conductor record by its hostname.

Parameters

- **cls** the *Conductor*
- **context** Security context
- **hostname** the hostname on which a Conductor is running
- **online** Specify the expected `online` field value for the conductor to be retrieved. The `online` field is ignored if this value is set to None.

Returns a *Conductor* object.

property hostname

property id

classmethod list(*context*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)
Return a list of Conductor objects.

Parameters

- **cls** the *Conductor*
- **context** Security context.
- **limit** maximum number of resources to return in a single result.
- **marker** pagination marker for large data sets.
- **sort_key** column to sort results by.
- **sort_dir** direction to sort. asc or desc.

Returns a list of *Conductor* object.

refresh(*context=None*)
Loads and applies updates for this Conductor.

Loads a *Conductor* with the same uuid from the database and checks for updated attributes. Updates are applied from the loaded chassis column by column, if there are any updates.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Conductor(context)

classmethod register(*context*, *hostname*, *drivers*, *conductor_group*, *update_existing=False*)
Register an active conductor with the cluster.

Parameters

- **cls** the *Conductor*
- **context** Security context
- **hostname** the hostname on which the conductor will run
- **drivers** the list of drivers enabled in the conductor
- **conductor_group** conductor group to join, used for node:conductor affinity.
- **update_existing** When false, registration will raise an exception when a conflicting online record is found. When true, will overwrite the existing record. Default: False.

Raises ConductorAlreadyRegistered

Returns a *Conductor* object.

register_hardware_interfaces(*interfaces*)
Register hardware interfaces with the conductor.

Parameters **interfaces** List of interface to register, each entry should be a dictionary conaining hardware_type, interface_type, interface_name and default, e.g. {hardware_type: hardware-type, interface_type: deploy, interface_name: direct, default: True}

save(*context*)
Save is not supported by Conductor objects.

touch(*context=None*)
Touch this conductors DB record, marking it as up-to-date.

unregister(*context=None*)
Remove this conductor from the service registry.

unregister_all_hardware_interfaces()
Unregister all hardware interfaces for this conductor.

property updated_at

ironic.objects.deploy_template module

class ironic.objects.deploy_template.**DeployTemplate**(*context=None*, ***kwargs*)
Bases: *ironic.objects.base.IronicObject*, oslo_versionedobjects.base.VersionedObjectDictCompat

VERSION = '1.1'

create(*context=None*)
Create a DeployTemplate record in the DB.

Parameters **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: DeployTemplate(context).

Raises DeployTemplateDuplicateName if a deploy template with the same name exists.

Raises DeployTemplateAlreadyExists if a deploy template with the same UUID exists.

property created_at

dbapi = <oslo_db.api.DBAPI object>

destroy()
Delete the DeployTemplate from the DB.

Parameters **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: DeployTemplate(context).

Raises DeployTemplateNotFound if the deploy template no longer appears in the database.

property extra

```
fields = {'created_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'extra': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'id':
Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False),
'name': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False),
'steps': List(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False),
'updated_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid':
UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False)}
```

classmethod get_by_id(*context*, *template_id*)
Find a deploy template based on its integer ID.

Parameters

- **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: DeployTemplate(context).
- **template_id** The ID of a deploy template.

Raises DeployTemplateNotFound if the deploy template no longer appears in the database.

Returns a *DeployTemplate* object.

classmethod get_by_name(*context*, *name*)
Find a deploy template based on its name.

Parameters

- **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: DeployTemplate(context).
- **name** The name of a deploy template.

Raises DeployTemplateNotFound if the deploy template no longer appears in the database.

Returns a *DeployTemplate* object.

classmethod get_by_uuid(*context*, *uuid*)
Find a deploy template based on its UUID.

Parameters

- **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: DeployTemplate(context).

- **uuid** The UUID of a deploy template.

Raises DeployTemplateNotFound if the deploy template no longer appears in the database.

Returns a *DeployTemplate* object.

property id

classmethod list(*context*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)
Return a list of DeployTemplate objects.

Parameters

- **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: DeployTemplate(context).
- **limit** maximum number of resources to return in a single result.
- **marker** pagination marker for large data sets.
- **sort_key** column to sort results by.
- **sort_dir** direction to sort. asc or desc.

Returns a list of *DeployTemplate* objects.

classmethod list_by_names(*context*, *names*)
Return a list of DeployTemplate objects matching a set of names.

Parameters

- **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: DeployTemplate(context).
- **names** a list of names to filter by.

Returns a list of *DeployTemplate* objects.

property name

refresh(*context=None*)
Loads updates for this deploy template.

Loads a deploy template with the same uuid from the database and checks for updated attributes. Updates are applied from the loaded template column by column, if there are any updates.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Port(context)

Raises DeployTemplateNotFound if the deploy template no longer appears in the database.

save(*context=None*)
Save updates to this DeployTemplate.

Column-wise updates will be made based on the result of self.what_changed().

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: DeployTemplate(context)

Raises DeployTemplateDuplicateName if a deploy template with the same name exists.

Raises DeployTemplateNotFound if the deploy template does not exist.

property steps

property updated_at

property uuid

class ironic.objects.deploy_template.**DeployTemplateCRUDNotification**(*context=None*, ***kwargs*)

Bases: *ironic.objects.notification.NotificationBase*

Notification emitted on deploy template API operations.

VERSION = '1.0'

property created_at

property event_type

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'event_type': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'level': NotificationLevel(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('debug', 'info', 'warning', 'error', 'critical')), 'payload': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'publisher': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property level

property payload

property publisher

property updated_at

class ironic.objects.deploy_template.**DeployTemplateCRUDPayload**(*deploy_template*, ***kwargs*)

Bases: *ironic.objects.notification.NotificationPayloadBase*

SCHEMA = {'created_at': ('deploy_template', 'created_at'), 'extra': ('deploy_template', 'extra'), 'name': ('deploy_template', 'name'), 'steps': ('deploy_template', 'steps'), 'updated_at': ('deploy_template', 'updated_at'), 'uuid': ('deploy_template', 'uuid')}

VERSION = '1.0'

property created_at

property extra

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'extra': FlexibleDict(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'name': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'steps': List(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid': UUID(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False)}

property name

property steps

property updated_at

property uuid

ironic.objects.deployment module

class ironic.objects.deployment.**Deployment**(*context=None*, ***kwargs*)

Bases: *ironic.objects.base.IronicObject*, oslo_versionedobjects.base.VersionedObjectDictCompat

VERSION = '1.0'

create(*context=None*, *node=None*)

Create a Deployment.

Updates the corresponding node under the hood.

Parameters

- **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Deployment(context)
- **node** Node object for deployment.

Raises InstanceAssociated, NodeAssociated, NodeNotFound

property created_at

dbapi = <oslo_db.api.DBAPI object>

destroy(*context=None*, *node=None*)

Delete the Deployment.

Updates the corresponding node under the hood.

Parameters

- **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Node(context)
- **node** Node object for deployment.

```
fields = {'created_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'image_checksum': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'image_ref': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'kernel_ref': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'node_uuid': UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'ramdisk_ref': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'root_device': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'root_gib': Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'state': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'swap_mib': Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'updated_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid':
UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}
```

classmethod get_by_node_uuid(*context*, *node_uuid*)

Find a deployment based by its nodes UUID.

Parameters

- **cls** the *Deployment*
- **context** Security context
- **node_uuid** The UUID of a corresponding node.

Returns An *Deployment* object.

Raises NodeNotFound

classmethod get_by_uuid(*context*, *uuid*)

Find a deployment by its UUID.

Parameters

- **cls** the *Deployment*
- **context** Security context
- **uuid** The UUID of a deployment.

Returns An *Deployment* object.

Raises InstanceNotFound

property image_checksum

property image_ref

instance_info_mapping = {'image_checksum': 'image_checksum', 'image_source': 'image_ref', 'kernel': 'kernel_ref', 'ramdisk': 'ramdisk_ref', 'root_device': 'root_device', 'root_gb': 'root_gib', 'swap_mb': 'swap_mib'}

instance_info_mapping_rev = {'image_checksum': 'image_checksum', 'image_ref': 'image_source', 'kernel_ref': 'kernel', 'ramdisk_ref': 'ramdisk', 'root_device': 'root_device', 'root_gib': 'root_gb', 'swap_mib': 'swap_mb'}

property kernel_ref

classmethod list(*context*, *filters=None*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)

Return a list of Deployment objects.

Parameters

- **cls** the *Deployment*
- **context** Security context.
- **filters** Filters to apply.
- **limit** Maximum number of resources to return in a single result.
- **marker** Pagination marker for large data sets.
- **sort_key** Column to sort results by.
- **sort_dir** Direction to sort. asc or desc.

Returns A list of *Deployment* object.

Raises InvalidParameterValue

node_mapping = {'instance_uuid': 'uuid', 'provision_state': 'state', 'uuid': 'node_uuid'}

property node_uuid

property ramdisk_ref

refresh(*context=None*)

Refresh the object by re-fetching from the DB.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Node(context)

property root_device

property root_gib

property state

property swap_mib

property updated_at

property uuid

ironic.objects.fields module

class ironic.objects.fields.**BooleanField**(***kwargs*)
Bases: oslo_versionedobjects.fields.BooleanField

class ironic.objects.fields.**DateTimeField**(*tzinfo_aware=True*, ***kwargs*)
Bases: oslo_versionedobjects.fields.DateTimeField

class ironic.objects.fields.**EnumField**(*valid_values*, ***kwargs*)
Bases: oslo_versionedobjects.fields.EnumField

class ironic.objects.fields.**FlexibleDict**
Bases: oslo_versionedobjects.fields.FieldType

static **coerce**(*obj*, *attr*, *value*)
This is called to coerce (if possible) a value on assignment.

This method should convert the value given into the designated type, or throw an exception if this is not possible.

Param:obj The VersionedObject on which an attribute is being set

Param:attr The name of the attribute being set

Param:value The value being set

Returns A properly-typed value

class ironic.objects.fields.**FlexibleDictField**(***kwargs*)
Bases: oslo_versionedobjects.fields.AutoTypedField

AUTO_TYPE = <ironic.objects.fields.FlexibleDict object>

class ironic.objects.fields.**IntegerField**(***kwargs*)
Bases: oslo_versionedobjects.fields.IntegerField

class ironic.objects.fields.**ListOfFlexibleDictsField**(***kwargs*)
Bases: oslo_versionedobjects.fields.AutoTypedField

AUTO_TYPE = <oslo_versionedobjects.fields.List object>

class ironic.objects.fields.**ListOfObjectsField**(*objtype*, *subclasses=False*, ***kwargs*)
Bases: oslo_versionedobjects.fields.ListOfObjectsField

class ironic.objects.fields.**ListOfStringsField**(***kwargs*)
Bases: oslo_versionedobjects.fields.ListOfStringsField

class ironic.objects.fields.**MACAddress**
Bases: oslo_versionedobjects.fields.FieldType

static **coerce**(*obj*, *attr*, *value*)
This is called to coerce (if possible) a value on assignment.

This method should convert the value given into the designated type, or throw an exception if this is not possible.

Param:obj The VersionedObject on which an attribute is being set

Param:attr The name of the attribute being set

Param:value The value being set

Returns A properly-typed value

class ironic.objects.fields.**MACAddressField**(***kwargs*)
Bases: oslo_versionedobjects.fields.AutoTypedField

AUTO_TYPE = <ironic.objects.fields.MACAddress object>

class ironic.objects.fields.**NotificationLevel**
Bases: oslo_versionedobjects.fields.Enum

ALL = ('debug', 'info', 'warning', 'error', 'critical')

CRITICAL = 'critical'

DEBUG = 'debug'

ERROR = 'error'

INFO = 'info'

WARNING = 'warning'

class ironic.objects.fields.**NotificationLevelField**(***kwargs*)
Bases: oslo_versionedobjects.fields.BaseEnumField

AUTO_TYPE = <ironic.objects.fields.NotificationLevel object>

class ironic.objects.fields.**NotificationStatus**
Bases: oslo_versionedobjects.fields.Enum

ALL = ('start', 'end', 'error', 'success')

END = 'end'

ERROR = 'error'

START = 'start'

SUCCESS = 'success'

class ironic.objects.fields.**NotificationStatusField**(***kwargs*)
Bases: oslo_versionedobjects.fields.BaseEnumField

AUTO_TYPE = <ironic.objects.fields.NotificationStatus object>

class ironic.objects.fields.**ObjectField**(*objtype*, *subclasses=False*, ***kwargs*)
Bases: oslo_versionedobjects.fields.ObjectField

class ironic.objects.fields.**StringAcceptsCallable**
Bases: oslo_versionedobjects.fields.String

static coerce(*obj*, *attr*, *value*)
This is called to coerce (if possible) a value on assignment.

This method should convert the value given into the designated type, or throw an exception if this is not possible.

Param:obj The VersionedObject on which an attribute is being set

Param:attr The name of the attribute being set

Param:value The value being set

Returns A properly-typed value

class ironic.objects.fields.**StringField**(***kwargs*)
Bases: oslo_versionedobjects.fields.StringField

class ironic.objects.fields.**StringFieldThatAcceptsCallable**(***kwargs*)
Bases: oslo_versionedobjects.fields.StringField

Custom StringField object that allows for functions as default

In some cases we need to allow for dynamic defaults based on configuration options, this StringField object allows for a function to be passed as a default, and will only process it at the point the field is coerced

AUTO_TYPE = <ironic.objects.fields.StringAcceptsCallable object>

class ironic.objects.fields.**UUIDField**(***kwargs*)
Bases: oslo_versionedobjects.fields.UUIDField

ironic.objects.indirection module

class ironic.objects.indirection.**IronicObjectIndirectionAPI**
Bases: oslo_versionedobjects.base.VersionedObjectIndirectionAPI

object_action(*context*, *objinst*, *objmethod*, *args*, *kwargs*)
Perform an action on a VersionedObject instance.

When indirection_api is set on a VersionedObject (to a class implementing this interface), method calls on remotable methods will cause this to be executed to actually make the desired call. This often involves performing RPC.

Parameters

- **context** The context within which to perform the action
- **objinst** The object instance on which to perform the action
- **objmethod** The name of the action method to call
- **args** The positional arguments to the action method
- **kwargs** The keyword arguments to the action method

Returns The result of the action method

object_backport_versions(*context*, *objinst*, *object_versions*)
Perform a backport of an object instance.

This method is basically just like object_backport() but instead of providing a specific target version for the toplevel object and relying on the service-side mapping to handle sub-objects, this sends a mapping of all the dependent objects and their client-supported versions. The server will backport objects within the tree starting at objinst to the versions specified in object_versions, removing objects that have no entry. Use obj_tree_get_versions() to generate this mapping.

NOTE: This was not in the initial spec for this interface, so the base class raises NotImplementedError if you dont implement it. For backports, this method will be tried first, and if unimplemented, will fall back to object_backport().

Parameters

- **context** The context within which to perform the backport
- **objinst** An instance of a VersionedObject to be backported
- **object_versions** A dict of {objname: version} mappings

object_class_action(*context*, *objname*, *objmethod*, *objver*, *args*, *kwargs*)

Deprecated since version 0.10.0.

Use *object_class_action_versions()* instead.

Perform an action on a VersionedObject class.

When indirection_api is set on a VersionedObject (to a class implementing this interface), classmethod calls on remotable_classmethod methods will cause this to be executed to actually make the desired call. This usually involves performing RPC.

Parameters

- **context** The context within which to perform the action
- **objname** The registry name of the object
- **objmethod** The name of the action method to call
- **objver** The (remote) version of the object on which the action is being taken
- **args** The positional arguments to the action method
- **kwargs** The keyword arguments to the action method

Returns The result of the action method, which may (or may not) be an instance of the implementing VersionedObject class.

object_class_action_versions(*context*, *objname*, *objmethod*, *object_versions*, *args*, *kwargs*)

Perform an action on a VersionedObject class.

When indirection_api is set on a VersionedObject (to a class implementing this interface), classmethod calls on remotable_classmethod methods will cause this to be executed to actually make the desired call. This usually involves performing RPC.

This differs from object_class_action() in that it is provided with object_versions, a manifest of client-side object versions for easier nested backports. The manifest is the result of calling obj_tree_get_versions().

NOTE: This was not in the initial spec for this interface, so the base class raises NotImplementedError if you dont implement it. For backports, this method will be tried first, and if unimplemented, will fall back to object_class_action(). New implementations should provide this method instead of object_class_action()

Parameters

- **context** The context within which to perform the action
- **objname** The registry name of the object
- **objmethod** The name of the action method to call
- **object_versions** A dict of {objname: version} mappings
- **args** The positional arguments to the action method

- **kwargs** The keyword arguments to the action method

Returns The result of the action method, which may (or may not) be an instance of the implementing VersionedObject class.

ironic.objects.node module

class ironic.objects.node.**Node**(*context=None*, ***kwargs*)

Bases: *ironic.objects.base.IronicObject*, oslo_versionedobjects.base.VersionedObjectDictCompat

VERSION = '1.36'

property allocation_id

as_dict(*secure=False*, *mask_configdrive=True*)

Return the object represented as a dict.

The returned object is JSON-serialisable.

property automated_clean

property bios_interface

property boot_interface

property boot_mode

property chassis_id

property clean_step

property conductor_affinity

property conductor_group

property console_enabled

property console_interface

create(*context=None*)

Create a Node record in the DB.

Column-wise updates will be made based on the result of self.what_changed(). If target_power_state is provided, it will be checked against the in-database copy of the node before updates are made.

Parameters context Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Node(context)

Raises InvalidParameterValue if some property values are invalid.

property created_at

dbapi = <oslo_db.api.DBAPI object>

del_driver_internal_info(*key*, *default_value=None*)

Pop a value from the driver_internal_info.

Removing a *driver_internal_info* dict value via this method ensures that this field will be flagged for saving.

Parameters

- **key** Key of item to pop off the *driver_internal_info* dict
- **default_value** Value to return if the key doesnt exist

Returns The removed value, or *default_value*

property deploy_interface

property deploy_step

property description

destroy(*context=None*)

Delete the Node from the DB.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Node(context)

property driver

property driver_info

property driver_internal_info

property extra

property fault

```
fields = {'allocation_id': Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'automated_clean': Boolean(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'bios_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'boot_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'boot_mode': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'chassis_id': Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'clean_step': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'conductor_affinity': Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'conductor_group': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'console_enabled': Boolean(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False),
'console_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'created_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'deploy_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'deploy_step': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'description': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'driver': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'driver_info': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'driver_internal_info': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'extra': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'fault': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'id':
Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False),
'inspect_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspection_finished_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspection_started_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'instance_info': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'instance_uuid': UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'last_error': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'lessee': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'maintenance': Boolean(default=<class
```

classmethod get(*context*, *node_id*)
Find a node based on its id or uuid and return a Node object.

Parameters

- **context** Security context
- **node_id** the id *or* uuid of a node.

Returns a *Node* object.

classmethod get_by_id(*context*, *node_id*)
Find a node based on its integer ID and return a Node object.

Parameters

- **cls** the *Node*
- **context** Security context
- **node_id** the ID of a node.

Returns a *Node* object.

classmethod get_by_instance_uuid(*context*, *instance_uuid*)
Find a node based on the instance UUID and return a Node object.

Parameters

- **cls** the *Node*
- **context** Security context
- **uuid** the UUID of the instance.

Returns a *Node* object.

classmethod get_by_name(*context*, *name*)
Find a node based on name and return a Node object.

Parameters

- **cls** the *Node*
- **context** Security context
- **name** the logical name of a node.

Returns a *Node* object.

classmethod get_by_port_addresses(*context*, *addresses*)
Get a node by associated port addresses.

Parameters

- **cls** the *Node*
- **context** Security context.
- **addresses** A list of port addresses.

Raises NodeNotFound if the node is not found.

Returns a *Node* object.

classmethod get_by_uuid(*context*, *uuid*)
Find a node based on UUID and return a Node object.

Parameters

- **cls** the *Node*
- **context** Security context
- **uuid** the UUID of a node.

Returns a *Node* object.

get_interface(*iface*)

property id

property inspect_interface

property inspection_finished_at

property inspection_started_at

property instance_info

property instance_uuid

property last_error

property lessee

classmethod list(*context*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *filters=None*, *fields=None*)
Return a list of Node objects.

Parameters

- **cls** the *Node*
- **context** Security context.
- **limit** maximum number of resources to return in a single result.
- **marker** pagination marker for large data sets.
- **sort_key** column to sort results by.
- **sort_dir** direction to sort. asc or desc.
- **filters** Filters to apply.
- **fields** Requested fields to be returned. Please note, some fields are mandatory for the data model and are automatically included. These are: id, version, updated_at, created_at, owner, and lessee.

Returns a list of *Node* object.

property maintenance

property maintenance_reason

property management_interface

property name

property network_data

property network_interface

property owner

property power_interface

property power_state

property properties

property protected

property protected_reason

property provision_state

property provision_updated_at

property raid_config

property raid_interface

refresh(*context=None*)

Refresh the object by re-fetching from the DB.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Node(context)

classmethod release(*context*, *tag*, *node_id*)

Release the reservation on a node.

Parameters

- **context** Security context.
- **tag** A string uniquely identifying the reservation holder.
- **node_id** A node id or uuid.

Raises NodeNotFound if the node is not found.

property rescue_interface

property reservation

classmethod reserve(*context*, *tag*, *node_id*)

Get and reserve a node.

To prevent other ManagerServices from manipulating the given Node while a Task is performed, mark it reserved by this host.

Parameters

- **cls** the *Node*
- **context** Security context.
- **tag** A string uniquely identifying the reservation holder.
- **node_id** A node ID or UUID.

Raises NodeNotFound if the node is not found.

Returns a *Node* object.

property resource_class

property retired

property retired_reason

save(*context=None*)
Save updates to this Node.

Column-wise updates will be made based on the result of self.what_changed(). If target_power_state is provided, it will be checked against the in-database copy of the node before updates are made.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Node(context)

Raises InvalidParameterValue if some property values are invalid.

property secure_boot

set_driver_internal_info(*key*, *value*)
Set a *driver_internal_info* value.

Setting a *driver_internal_info* dict value via this method ensures that this field will be flagged for saving.

Parameters

- **key** Key of item to set
- **value** Value of item to set

set_instance_info(*key*, *value*)
Set an *instance_info* value.

Setting a *instance_info* dict value via this method ensures that this field will be flagged for saving.

Parameters

- **key** Key of item to set
- **value** Value of item to set

property storage_interface

property target_power_state

property target_provision_state

property target_raid_config

timestamp_driver_internal_info(*key*)
Set a *driver_internal_info* value with the current timestamp.

Setting a *driver_internal_info* timestamp value via this method ensures that this field will be flagged for saving.

Parameters **key** Key of item to set the timestamp on

touch_provisioning(*context=None*)
Touch the database record to mark the provisioning as alive.

property traits

property updated_at

property uuid

property vendor_interface

class ironic.objects.node.**NodeCRUDNotification**(*context=None*, ***kwargs*)

Bases: *ironic.objects.notification.NotificationBase*

Notification emitted when ironic creates, updates or deletes a node.

VERSION = '1.0'

property created_at

property event_type

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'event_type': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'level': NotificationLevel(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('debug', 'info', 'warning', 'error', 'critical')), 'payload': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'publisher': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property level

property payload

property publisher

property updated_at

class ironic.objects.node.**NodeCRUDPayload**(*node*, *chassis_uuid*)

Bases: *ironic.objects.node.NodePayload*

Payload schema for when ironic creates, updates or deletes a node.

SCHEMA = {'bios_interface': ('node', 'bios_interface'), 'boot_interface': ('node', 'boot_interface'), 'boot_mode': ('node', 'boot_mode'), 'clean_step': ('node', 'clean_step'), 'conductor_group': ('node', 'conductor_group'), 'console_enabled': ('node', 'console_enabled'), 'console_interface': ('node', 'console_interface'), 'created_at': ('node', 'created_at'), 'deploy_interface': ('node', 'deploy_interface'), 'deploy_step': ('node', 'deploy_step'), 'description': ('node', 'description'), 'driver': ('node', 'driver'), 'driver_info': ('node', 'driver_info'), 'extra': ('node', 'extra'), 'fault': ('node', 'fault'), 'inspect_interface': ('node', 'inspect_interface'), 'inspection_finished_at': ('node', 'inspection_finished_at'), 'inspection_started_at': ('node', 'inspection_started_at'), 'instance_info': ('node', 'instance_info'), 'instance_uuid': ('node', 'instance_uuid'), 'last_error': ('node', 'last_error'), 'lessee': ('node', 'lessee'), 'maintenance': ('node', 'maintenance'), 'maintenance_reason': ('node', 'maintenance_reason'), 'management_interface': ('node', 'management_interface'), 'name': ('node', 'name'), 'network_interface': ('node', 'network_interface'), 'owner': ('node', 'owner'), 'power_interface': ('node', 'power_interface'), 'power_state': ('node', 'power_state'), 'properties': ('node', 'properties'), 'protected': ('node', 'protected'), 'protected_reason': ('node', 'protected_reason'), 'provision_state': ('node', 'provision_state'), 'provision_updated_at': ('node', 'provision_updated_at'), 'raid_interface': ('node', 'raid_interface'), 'rescue_interface': ('node', 'rescue_interface'), 'resource_class': ('node', 'resource_class'), 'retired': ('node', 'retired'), 'retired_reason': ('node', 'retired_reason'), 'secure_boot': ('node', 'secure_boot'), 'storage_interface': ('node', 'storage_interface'), 'target_power_state': ('node', 'target_power_state'), 'target_provision_state': ('node', 'target_provision_state'), 'updated_at': ('node', 'updated_at'), 'uuid': ('node', 'uuid'), 'vendor_interface': ('node', 'vendor_interface')}

VERSION = '1.14'

property bios_interface

property boot_interface

property boot_mode

property chassis_uuid

property clean_step

property conductor_group

property console_enabled

property console_interface

property created_at

property deploy_interface

property deploy_step

property description

```
property driver
property driver_info
property extra
property fault
```

```
fields = {'bios_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'boot_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'boot_mode': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'chassis_uuid': UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'clean_step': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'conductor_group': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'console_enabled': Boolean(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'console_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'created_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'deploy_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'deploy_step': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'description': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'driver': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'driver_info': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'extra': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'fault': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspect_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspection_finished_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspection_started_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'instance_info': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'instance_uuid': UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'last_error': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'lessee': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'maintenance': Boolean(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'maintenance_reason': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'management_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'name':
String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'network_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'owner': String(default=<class
```

property inspect_interface

property inspection_finished_at

property inspection_started_at

property instance_info

property instance_uuid

property last_error

property lessee

property maintenance

property maintenance_reason

property management_interface

property name

property network_interface

property owner

property power_interface

property power_state

property properties

property protected

property protected_reason

property provision_state

property provision_updated_at

property raid_interface

property rescue_interface

property resource_class

property retired

property retired_reason

property secure_boot

property storage_interface

property target_power_state

property target_provision_state

property traits

property updated_at

property uuid

property vendor_interface

class ironic.objects.node.**NodeConsoleNotification**(*context=None*, ***kwargs*)

Bases: *ironic.objects.notification.NotificationBase*

Notification emitted when node console state changed.

VERSION = '1.0'

property created_at

property event_type

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'event_type': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'level': NotificationLevel(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('debug', 'info', 'warning', 'error', 'critical')), 'payload': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'publisher': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property level

property payload

property publisher

property updated_at

class ironic.objects.node.**NodeCorrectedPowerStateNotification**(*context=None*, ***kwargs*)

Bases: *ironic.objects.notification.NotificationBase*

Notification for when a nodes power state is corrected in the database.

This notification is emitted when ironic detects that the actual power state on a bare metal hardware is different from the power state on an ironic node (DB). This notification is emitted after the database is updated to reflect this correction.

VERSION = '1.0'

property created_at

property event_type

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'event_type': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'level': NotificationLevel(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('debug', 'info', 'warning', 'error', 'critical')), 'payload': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'publisher': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property level

property payload

property publisher

property updated_at

class ironic.objects.node.**NodeCorrectedPowerStatePayload**(*node*, *from_power*)

Bases: *ironic.objects.node.NodePayload*

Notification payload schema for when a nodes power state is corrected.

from_power indicates the previous power state on the ironic node before the node was updated.

VERSION = '1.16'

property bios_interface

property boot_interface

property boot_mode

property clean_step

property conductor_group

property console_enabled

property console_interface

property created_at

property deploy_interface

property deploy_step

property description

property driver

property extra

property fault

```
fields = {'bios_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'boot_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'boot_mode': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'clean_step': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'conductor_group': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'console_enabled': Boolean(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'console_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'created_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'deploy_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'deploy_step': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'description': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'driver': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'extra': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'fault': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'from_power': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspect_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspection_finished_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspection_started_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'instance_uuid': UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'last_error': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'lessee': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'maintenance': Boolean(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'maintenance_reason': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'management_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'name':
String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'network_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'owner': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'power_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'power_state': String(default=<class
```

property from_power

property inspect_interface

property inspection_finished_at

property inspection_started_at

property instance_uuid

property last_error

property lessee

property maintenance

property maintenance_reason

property management_interface

property name

property network_interface

property owner

property power_interface

property power_state

property properties

property protected

property protected_reason

property provision_state

property provision_updated_at

property raid_interface

property rescue_interface

property resource_class

property retired

property retired_reason

property secure_boot

property storage_interface

property target_power_state

property target_provision_state

property traits

property updated_at

property uuid

property vendor_interface

class ironic.objects.node.**NodeMaintenanceNotification**(*context=None*, ***kwargs*)

Bases: *ironic.objects.notification.NotificationBase*

Notification emitted when maintenance state changed via API.

VERSION = '1.0'

property created_at

property event_type

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'event_type': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'level': NotificationLevel(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('debug', 'info', 'warning', 'error', 'critical')), 'payload': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'publisher': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property level

property payload

property publisher

property updated_at

class ironic.objects.node.**NodePayload**(*node*, ***kwargs*)

Bases: *ironic.objects.notification.NotificationPayloadBase*

Base class used for all notification payloads about a Node object.

SCHEMA = {'bios_interface': ('node', 'bios_interface'), 'boot_interface': ('node', 'boot_interface'), 'boot_mode': ('node', 'boot_mode'), 'clean_step': ('node', 'clean_step'), 'conductor_group': ('node', 'conductor_group'), 'console_enabled': ('node', 'console_enabled'), 'console_interface': ('node', 'console_interface'), 'created_at': ('node', 'created_at'), 'deploy_interface': ('node', 'deploy_interface'), 'deploy_step': ('node', 'deploy_step'), 'description': ('node', 'description'), 'driver': ('node', 'driver'), 'extra': ('node', 'extra'), 'fault': ('node', 'fault'), 'inspect_interface': ('node', 'inspect_interface'), 'inspection_finished_at': ('node', 'inspection_finished_at'), 'inspection_started_at': ('node', 'inspection_started_at'), 'instance_uuid': ('node', 'instance_uuid'), 'last_error': ('node', 'last_error'), 'lessee': ('node', 'lessee'), 'maintenance': ('node', 'maintenance'), 'maintenance_reason': ('node', 'maintenance_reason'), 'management_interface': ('node', 'management_interface'), 'name': ('node', 'name'), 'network_interface': ('node', 'network_interface'), 'owner': ('node', 'owner'), 'power_interface': ('node', 'power_interface'), 'power_state': ('node', 'power_state'), 'properties': ('node', 'properties'), 'protected': ('node', 'protected'), 'protected_reason': ('node', 'protected_reason'), 'provision_state': ('node', 'provision_state'), 'provision_updated_at': ('node', 'provision_updated_at'), 'raid_interface': ('node', 'raid_interface'), 'rescue_interface': ('node', 'rescue_interface'), 'resource_class': ('node', 'resource_class'), 'retired': ('node', 'retired'), 'retired_reason': ('node', 'retired_reason'), 'secure_boot': ('node', 'secure_boot'), 'storage_interface': ('node', 'storage_interface'), 'target_power_state': ('node', 'target_power_state'), 'target_provision_state': ('node', 'target_provision_state'), 'updated_at': ('node', 'updated_at'), 'uuid': ('node', 'uuid'), 'vendor_interface': ('node', 'vendor_interface')}

VERSION = '1.16'

property bios_interface

property boot_interface

property boot_mode

property clean_step

property conductor_group

property console_enabled

property console_interface

property created_at

property deploy_interface

property deploy_step

property description

property driver

property extra

```
property fault
```

```
fields = {'bios_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'boot_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'boot_mode': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'clean_step': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'conductor_group': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'console_enabled': Boolean(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'console_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'created_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'deploy_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'deploy_step': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'description': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'driver': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'extra': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'fault': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspect_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspection_finished_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspection_started_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'instance_uuid': UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'last_error': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'lessee': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'maintenance': Boolean(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'maintenance_reason': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'management_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'name':
String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'network_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'owner': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'power_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'power_state': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'properties': FlexibleDict(default=<class
```

property inspect_interface

property inspection_finished_at

property inspection_started_at

property instance_uuid

property last_error

property lessee

property maintenance

property maintenance_reason

property management_interface

property name

property network_interface

property owner

property power_interface

property power_state

property properties

property protected

property protected_reason

property provision_state

property provision_updated_at

property raid_interface

property rescue_interface

property resource_class

property retired

property retired_reason

property secure_boot

property storage_interface

property target_power_state

property target_provision_state

property traits

property updated_at

property uuid

property vendor_interface

class ironic.objects.node.**NodeSetPowerStateNotification**(*context=None*, ***kwargs*)

Bases: *ironic.objects.notification.NotificationBase*

Notification emitted when ironic changes a nodes power state.

VERSION = '1.0'

property created_at

property event_type

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'event_type': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'level': NotificationLevel(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('debug', 'info', 'warning', 'error', 'critical')), 'payload': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'publisher': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property level

property payload

property publisher

property updated_at

class ironic.objects.node.**NodeSetPowerStatePayload**(*node*, *to_power*)

Bases: *ironic.objects.node.NodePayload*

Payload schema for when ironic changes a nodes power state.

VERSION = '1.16'

property bios_interface

property boot_interface

property boot_mode

property clean_step

property conductor_group

property console_enabled

property console_interface

property created_at

property deploy_interface

property deploy_step

property description

property driver

property extra

property fault

```
fields = {'bios_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'boot_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'boot_mode': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'clean_step': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'conductor_group': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'console_enabled': Boolean(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'console_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'created_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'deploy_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'deploy_step': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'description': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'driver': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'extra': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'fault': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspect_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspection_finished_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspection_started_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'instance_uuid': UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'last_error': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'lessee': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'maintenance': Boolean(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'maintenance_reason': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'management_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'name':
String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'network_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'owner': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'power_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'power_state': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'properties': FlexibleDict(default=<class
```

property inspect_interface

property inspection_finished_at

property inspection_started_at

property instance_uuid

property last_error

property lessee

property maintenance

property maintenance_reason

property management_interface

property name

property network_interface

property owner

property power_interface

property power_state

property properties

property protected

property protected_reason

property provision_state

property provision_updated_at

property raid_interface

property rescue_interface

property resource_class

property retired

property retired_reason

property secure_boot

property storage_interface

property target_power_state

property target_provision_state

property to_power

property traits

property updated_at

property uuid

property vendor_interface

class ironic.objects.node.**NodeSetProvisionStateNotification**(*context=None*, ***kwargs*)

Bases: *ironic.objects.notification.NotificationBase*

Notification emitted when ironic changes a node provision state.

VERSION = '1.0'

property created_at

property event_type

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'event_type': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'level': NotificationLevel(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('debug', 'info', 'warning', 'error', 'critical')), 'payload': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'publisher': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property level

property payload

property publisher

property updated_at

class ironic.objects.node.**NodeSetProvisionStatePayload**(*node*, *prev_state*, *prev_target*, *event*)

Bases: *ironic.objects.node.NodePayload*

Payload schema for when ironic changes a node provision state.

SCHEMA = {'bios_interface': ('node', 'bios_interface'), 'boot_interface': ('node', 'boot_interface'), 'boot_mode': ('node', 'boot_mode'), 'clean_step': ('node', 'clean_step'), 'conductor_group': ('node', 'conductor_group'), 'console_enabled': ('node', 'console_enabled'), 'console_interface': ('node', 'console_interface'), 'created_at': ('node', 'created_at'), 'deploy_interface': ('node', 'deploy_interface'), 'deploy_step': ('node', 'deploy_step'), 'description': ('node', 'description'), 'driver': ('node', 'driver'), 'driver_internal_info': ('node', 'driver_internal_info'), 'extra': ('node', 'extra'), 'fault': ('node', 'fault'), 'inspect_interface': ('node', 'inspect_interface'), 'inspection_finished_at': ('node', 'inspection_finished_at'), 'inspection_started_at': ('node', 'inspection_started_at'), 'instance_info': ('node', 'instance_info'), 'instance_uuid': ('node', 'instance_uuid'), 'last_error': ('node', 'last_error'), 'lessee': ('node', 'lessee'), 'maintenance': ('node', 'maintenance'), 'maintenance_reason': ('node', 'maintenance_reason'), 'management_interface': ('node', 'management_interface'), 'name': ('node', 'name'), 'network_interface': ('node', 'network_interface'), 'owner': ('node', 'owner'), 'power_interface': ('node', 'power_interface'), 'power_state': ('node', 'power_state'), 'properties': ('node', 'properties'), 'protected': ('node', 'protected'), 'protected_reason': ('node', 'protected_reason'), 'provision_state': ('node', 'provision_state'), 'provision_updated_at': ('node', 'provision_updated_at'), 'raid_interface': ('node', 'raid_interface'), 'rescue_interface': ('node', 'rescue_interface'), 'resource_class': ('node', 'resource_class'), 'retired': ('node', 'retired'), 'retired_reason': ('node', 'retired_reason'), 'secure_boot': ('node', 'secure_boot'), 'storage_interface': ('node', 'storage_interface'), 'target_power_state': ('node', 'target_power_state'), 'target_provision_state': ('node', 'target_provision_state'), 'updated_at': ('node', 'updated_at'), 'uuid': ('node', 'uuid'), 'vendor_interface': ('node', 'vendor_interface')}

VERSION = '1.17'

property bios_interface

property boot_interface

property boot_mode

property clean_step

property conductor_group

property console_enabled

property console_interface

property created_at

property deploy_interface

property deploy_step

property description

property driver

```
property driver_internal_info
property event
property extra
property fault
```

```
fields = {'bios_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'boot_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'boot_mode': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'clean_step': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'conductor_group': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'console_enabled': Boolean(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'console_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'created_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'deploy_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'deploy_step': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'description': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'driver': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'driver_internal_info': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'event': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'extra': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'fault': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspect_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspection_finished_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'inspection_started_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'instance_info': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'instance_uuid': UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'last_error': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'lessee': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'maintenance': Boolean(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'maintenance_reason': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'management_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'name':
String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'network_interface': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'owner': String(default=<class
```

property inspect_interface

property inspection_finished_at

property inspection_started_at

property instance_info

property instance_uuid

property last_error

property lessee

property maintenance

property maintenance_reason

property management_interface

property name

property network_interface

property owner

property power_interface

property power_state

property previous_provision_state

property previous_target_provision_state

property properties

property protected

property protected_reason

property provision_state

property provision_updated_at

property raid_interface

property rescue_interface

property resource_class

property retired

property retired_reason

property secure_boot

property storage_interface

property target_power_state

property target_provision_state

property traits

property updated_at

property uuid

property vendor_interface

ironic.objects.node_history module

class ironic.objects.node_history.**NodeHistory**(*context=None*, ***kwargs*)

Bases: *ironic.objects.base.IronicObject*, oslo_versionedobjects.base.VersionedObjectDictCompat

VERSION = '1.0'

property conductor

create(*context=None*)

Create a NodeHistory record in the DB.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: NodeHistory(context)

property created_at

dbapi = <oslo_db.api.DBAPI object>

destroy(*context=None*)

Delete the NodeHistory from the DB.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: NodeHistory(context)

Raises NodeHistoryNotFound

property event

property event_type

```
fields = {'conductor': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'created_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'event': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'event_type': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'id':
Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False),
'node_id': Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'severity': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'updated_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'user':
String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid':
UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}
```

classmethod get(*context*, *history_ident*)
Get a history based on its id or uuid.

Parameters

- **history_ident** The id or uuid of a history.
- **context** Security context

Returns A *NodeHistory* object.

Raises InvalidIdentity

classmethod get_by_id(*context*, *history_id*)
Get a NodeHistory object by its integer ID.

Parameters

- **cls** the *NodeHistory*
- **context** Security context
- **history_id** The ID of a history.

Returns A *NodeHistory* object.

Raises NodeHistoryNotFound

classmethod get_by_uuid(*context*, *uuid*)
Get a NodeHistory object by its UUID.

Parameters

- **cls** the *NodeHistory*
- **context** Security context
- **uuid** The UUID of a NodeHistory.

Returns A *NodeHistory* object.

Raises NodeHistoryNotFound

property id

classmethod list(*context*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)
Return a list of NodeHistory objects.

Parameters

- **cls** the *NodeHistory*
- **context** Security context.
- **limit** Maximum number of resources to return in a single result.
- **marker** Pagination marker for large data sets.
- **sort_key** Column to sort results by.
- **sort_dir** Direction to sort. asc or desc.

Returns A list of *NodeHistory* object.

Raises InvalidParameterValue

classmethod list_by_node_id(*context*, *node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*)
Return a list of NodeHistory objects belongs to a given node ID.

Parameters

- **cls** the *NodeHistory*
- **context** Security context.
- **node_id** The ID of the node.
- **limit** Maximum number of resources to return in a single result.
- **marker** Pagination marker for large data sets.
- **sort_key** Column to sort results by.
- **sort_dir** Direction to sort. asc or desc.

Returns A list of *NodeHistory* object.

Raises InvalidParameterValue

property node_id

property severity

property updated_at

property user

property uuid

ironic.objects.notification module

class ironic.objects.notification.**EventType**(*context=None*, ***kwargs*)
Bases: *ironic.objects.base.IronicObject*

Defines the event_type to be sent on the wire.

An EventType must specify the object being acted on, a string describing the action being taken on the notification, and the status of the action.

VERSION = '1.1'

property action

property created_at

fields = {'action': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'object': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'status': NotificationStatus(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('start', 'end', 'error', 'success')), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property object

property status

to_event_type_field()
Constructs string for event_type to be sent on the wire.

The string is in the format: baremetal.<object>.<action>.<status>

Raises ValueError if self.status is not one of `fields.NotificationStatusField`

Returns event_type string

property updated_at

class ironic.objects.notification.**NotificationBase**(*context=None*, ***kwargs*)
Bases: *ironic.objects.base.IronicObject*

Base class for versioned notifications.

Subclasses must define the payload field, which must be a subclass of NotificationPayloadBase.

VERSION = '1.0'

property created_at

emit(*context*)
Send the notification.

Raises NotificationPayloadError

Raises oslo_versionedobjects.exceptions.MessageDeliveryFailure

property event_type

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'event_type': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'level': NotificationLevel(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('debug', 'info', 'warning', 'error', 'critical')), 'publisher': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property level

property publisher

property updated_at

class ironic.objects.notification.NotificationPayloadBase(*args, **kwargs)

Bases: *ironic.objects.base.IronicObject*

Base class for the payload of versioned notifications.

SCHEMA = {}

VERSION = '1.0'

property created_at

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

populate_schema(**kwargs)

Populate the object based on the SCHEMA and the source objects

Parameters **kwargs** A dict contains the source object and the keys defined in the SCHEMA

Raises NotificationSchemaObjectError

Raises NotificationSchemaKeyError

property updated_at

class ironic.objects.notification.NotificationPublisher(context=None, **kwargs)

Bases: *ironic.objects.base.IronicObject*

VERSION = '1.0'

property created_at

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'host': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'service': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property host

property service

property updated_at

ironic.objects.notification.**mask_secrets**(*payload*)
Remove secrets from payload object.

ironic.objects.port module

class ironic.objects.port.**Port**(*context=None*, ***kwargs*)
Bases: *ironic.objects.base.IronicObject*, oslo_versionedobjects.base.VersionedObjectDictCompat

VERSION = '1.10'

property address

create(*context=None*)
Create a Port record in the DB.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Port(context)

Raises MACAlreadyExists if address column is not unique

Raises PortAlreadyExists if uuid column is not unique

property created_at

dbapi = <oslo_db.api.DBAPI object>

destroy(*context=None*)
Delete the Port from the DB.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Port(context)

Raises PortNotFound

property extra

```
fields = {'address': MACAddress(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'created_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'extra': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'id':
Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False),
'internal_info': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'is_smartnic': Boolean(default=False,nullable=True),
'local_link_connection': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'name':
String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'node_id': Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'physical_network': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'portgroup_id': Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'pxe_enabled': Boolean(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False),
'updated_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid':
UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}
```

classmethod get(*context*, *port_id*)

Find a port.

Find a port based on its id or uuid or name or MAC address and return a Port object.

Parameters

- **context** Security context
- **port_id** the id *or* uuid *or* name *or* MAC address of a port.

Returns a *Port* object.

Raises InvalidIdentity

classmethod get_by_address(*context*, *address*, *owner=None*, *project=None*)

Find a port based on address and return a *Port* object.

Parameters

- **cls** the *Port*
- **context** Security context
- **address** the address of a port.
- **owner** DEPRECATED a node owner to match against
- **project** a node owner or lessee to match against

Returns a *Port* object.

Raises PortNotFound

classmethod get_by_id(*context*, *port_id*)
Find a port based on its integer ID and return a Port object.

Parameters

- **cls** the *Port*
- **context** Security context
- **port_id** the ID of a port.

Returns a *Port* object.

Raises PortNotFound

classmethod get_by_name(*context*, *name*)
Find a port based on name and return a *Port* object.

Parameters

- **cls** the *Port*
- **context** Security context
- **name** the name of a port.

Returns a *Port* object.

Raises PortNotFound

classmethod get_by_uuid(*context*, *uuid*)
Find a port based on UUID and return a *Port* object.

Parameters

- **cls** the *Port*
- **context** Security context
- **uuid** the UUID of a port.

Returns a *Port* object.

Raises PortNotFound

property id

property internal_info

property is_smartnic

classmethod list(*context*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *owner=None*, *project=None*)
Return a list of Port objects.

Parameters

- **context** Security context.
- **limit** maximum number of resources to return in a single result.
- **marker** pagination marker for large data sets.
- **sort_key** column to sort results by.

- **sort_dir** direction to sort. asc or desc.
- **owner** DEPRECATED a node owner to match against
- **project** a node owner or lessee to match against

Returns a list of *Port* object.

Raises InvalidParameterValue

classmethod list_by_node_id(*context*, *node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *owner=None*, *project=None*)

Return a list of Port objects associated with a given node ID.

Parameters

- **context** Security context.
- **node_id** the ID of the node.
- **limit** maximum number of resources to return in a single result.
- **marker** pagination marker for large data sets.
- **sort_key** column to sort results by.
- **sort_dir** direction to sort. asc or desc.
- **owner** DEPRECATED a node owner to match against
- **project** a node owner or lessee to match against

Returns a list of *Port* object.

classmethod list_by_portgroup_id(*context*, *portgroup_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *owner=None*, *project=None*)

Return a list of Port objects associated with a given portgroup ID.

Parameters

- **context** Security context.
- **portgroup_id** the ID of the portgroup.
- **limit** maximum number of resources to return in a single result.
- **marker** pagination marker for large data sets.
- **sort_key** column to sort results by.
- **sort_dir** direction to sort. asc or desc.
- **owner** DEPRECATED a node owner to match against
- **project** a node owner or lessee to match against

Returns a list of *Port* object.

property local_link_connection

property name

property node_id

property physical_network

property portgroup_id

property pxe_enabled

refresh(*context=None*)
Loads updates for this Port.

Loads a port with the same uuid from the database and checks for updated attributes. Updates are applied from the loaded port column by column, if there are any updates.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Port(context)

Raises PortNotFound

save(*context=None*)
Save updates to this Port.

Updates will be made column by column based on the result of self.what_changed().

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Port(context)

Raises PortNotFound

Raises MACAlreadyExists if address column is not unique

classmethod supports_is_smartnic()
Return whether is_smartnic field is supported.

Returns Whether is_smartnic field is supported

Raises ovo_exception.IncompatibleObjectVersion

classmethod supports_physical_network()
Return whether the physical_network field is supported.

Returns Whether the physical_network field is supported

Raises ovo_exception.IncompatibleObjectVersion

property updated_at

property uuid

class ironic.objects.port.**PortCRUDNotification**(*context=None*, ***kwargs*)
Bases: *ironic.objects.notification.NotificationBase*

Notification emitted when ironic creates, updates or deletes a port.

VERSION = '1.0'

property created_at

property event_type

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'event_type': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'level': NotificationLevel(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('debug', 'info', 'warning', 'error', 'critical')), 'payload': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'publisher': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property level

property payload

property publisher

property updated_at

class ironic.objects.port.PortCRUDPayload(*port*, *node_uuid*, *portgroup_uuid*)

Bases: *ironic.objects.notification.NotificationPayloadBase*

SCHEMA = {'address': ('port', 'address'), 'created_at': ('port', 'created_at'), 'extra': ('port', 'extra'), 'is_smartnic': ('port', 'is_smartnic'), 'local_link_connection': ('port', 'local_link_connection'), 'name': ('port', 'name'), 'physical_network': ('port', 'physical_network'), 'pxe_enabled': ('port', 'pxe_enabled'), 'updated_at': ('port', 'updated_at'), 'uuid': ('port', 'uuid')}

VERSION = '1.4'

property address

property created_at

property extra

fields = {'address': MACAddress(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'extra': FlexibleDict(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'is_smartnic': Boolean(default=False,nullable=True), 'local_link_connection': FlexibleDict(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'name': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'node_uuid': UUID(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'physical_network': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'portgroup_uuid': UUID(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'pxe_enabled': Boolean(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid': UUID(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False)}

property is_smartnic

property local_link_connection

property name

property node_uuid

property physical_network

property portgroup_uuid

property pxe_enabled

property updated_at

property uuid

ironic.objects.portgroup module

class ironic.objects.portgroup.**Portgroup**(*context=None*, ***kwargs*)

Bases: *ironic.objects.base.IronicObject*, oslo_versionedobjects.base.VersionedObjectDictCompat

VERSION = '1.4'

property address

create(*context=None*)

Create a Portgroup record in the DB.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first

argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Portgroup(context)

Raises DuplicateName, MACAlreadyExists, PortgroupAlreadyExists

property created_at

dbapi = <oslo_db.api.DBAPI object>

destroy(*context=None*)

Delete the Portgroup from the DB.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Portgroup(context)

Raises PortgroupNotEmpty, PortgroupNotFound

property extra

fields = {'address': MACAddress(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'extra': FlexibleDict(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'id': Integer(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'internal_info': FlexibleDict(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'mode': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'name': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'node_id': Integer(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'properties': FlexibleDict(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'standalone_ports_supported': Boolean(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid': UUID(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

classmethod get(*context*, *portgroup_ident*)

Find a portgroup based on its id, uuid, name or address.

Parameters

- **portgroup_ident** The id, uuid, name or address of a portgroup.
- **context** Security context

Returns A *Portgroup* object.

Raises InvalidIdentity

classmethod get_by_address(*context*, *address*, *project=None*)
Find portgroup by address and return a *Portgroup* object.

Parameters

- **cls** the *Portgroup*
- **context** Security context
- **address** The MAC address of a portgroup.
- **project** a node owner or lessee to match against.

Returns A *Portgroup* object.

Raises PortgroupNotFound

classmethod get_by_id(*context*, *portgroup_id*)
Find a portgroup by its integer ID and return a Portgroup object.

Parameters

- **cls** the *Portgroup*
- **context** Security context
- **portgroup_id** The ID of a portgroup.

Returns A *Portgroup* object.

Raises PortgroupNotFound

classmethod get_by_name(*context*, *name*)
Find portgroup based on name and return a *Portgroup* object.

Parameters

- **cls** the *Portgroup*
- **context** Security context
- **name** The name of a portgroup.

Returns A *Portgroup* object.

Raises PortgroupNotFound

classmethod get_by_uuid(*context*, *uuid*)
Find a portgroup by UUID and return a *Portgroup* object.

Parameters

- **cls** the *Portgroup*
- **context** Security context
- **uuid** The UUID of a portgroup.

Returns A *Portgroup* object.

Raises PortgroupNotFound

property id

property internal_info

classmethod list(*context*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)

Return a list of Portgroup objects.

Parameters

- **cls** the *Portgroup*
- **context** Security context.
- **limit** Maximum number of resources to return in a single result.
- **marker** Pagination marker for large data sets.
- **sort_key** Column to sort results by.
- **sort_dir** Direction to sort. asc or desc.
- **project** a node owner or lessee to match against.

Returns A list of *Portgroup* object.

Raises InvalidParameterValue

classmethod list_by_node_id(*context*, *node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)

Return a list of Portgroup objects associated with a given node ID.

Parameters

- **cls** the *Portgroup*
- **context** Security context.
- **node_id** The ID of the node.
- **limit** Maximum number of resources to return in a single result.
- **marker** Pagination marker for large data sets.
- **sort_key** Column to sort results by.
- **sort_dir** Direction to sort. asc or desc.
- **project** a node owner or lessee to match against.

Returns A list of *Portgroup* object.

Raises InvalidParameterValue

property mode

property name

property node_id

property properties

refresh(*context=None*)

Loads updates for this Portgroup.

Loads a portgroup with the same uuid from the database and checks for updated attributes. Updates are applied from the loaded portgroup column by column, if there are any updates.

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Portgroup(context)

Raises PortgroupNotFound

save(*context=None*)
Save updates to this Portgroup.

Updates will be made column by column based on the result of self.what_changed().

Parameters **context** Security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Portgroup(context)

Raises PortgroupNotFound, DuplicateName, MACAlreadyExists

property standalone_ports_supported

property updated_at

property uuid

class ironic.objects.portgroup.**PortgroupCRUDNotification**(*context=None*, ***kwargs*)
Bases: *ironic.objects.notification.NotificationBase*

Notification when ironic creates, updates or deletes a portgroup.

VERSION = '1.0'

property created_at

property event_type

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'event_type': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'level': NotificationLevel(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('debug', 'info', 'warning', 'error', 'critical')), 'payload': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'publisher': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property level

property payload

property publisher

property updated_at

class ironic.objects.portgroup.**PortgroupCRUDPayload**(*portgroup*, *node_uuid*)
Bases: *ironic.objects.notification.NotificationPayloadBase*

SCHEMA = {'address': ('portgroup', 'address'), 'created_at': ('portgroup', 'created_at'), 'extra': ('portgroup', 'extra'), 'mode': ('portgroup', 'mode'), 'name': ('portgroup', 'name'), 'properties': ('portgroup', 'properties'), 'standalone_ports_supported': ('portgroup', 'standalone_ports_supported'), 'updated_at': ('portgroup', 'updated_at'), 'uuid': ('portgroup', 'uuid')}

VERSION = '1.0'

property address

property created_at

property extra

fields = {'address': MACAddress(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'extra': FlexibleDict(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'mode': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'name': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'node_uuid': UUID(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'properties': FlexibleDict(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'standalone_ports_supported': Boolean(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid': UUID(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False)}

property mode

property name

property node_uuid

property properties

property standalone_ports_supported

property updated_at

property uuid

ironic.objects.trait module

class ironic.objects.trait.**Trait**(*context=None*, ***kwargs*)
Bases: *ironic.objects.base.IronicObject*

VERSION = '1.0'

create(*context=None*)
Create a Trait record in the DB.

Parameters **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Trait(context).

Raises InvalidParameterValue if adding the trait would exceed the per-node traits limit.

Raises NodeNotFound if the node no longer appears in the database.

property created_at

dbapi = <oslo_db.api.DBAPI object>

classmethod destroy(*context*, *node_id*, *trait*)
Delete the Trait from the DB.

Parameters

- **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Trait(context).
- **node_id** The id of a node.
- **trait** A trait string.

Raises NodeNotFound if the node no longer appears in the database.

Raises NodeTraitNotFound if the trait is not found.

classmethod exists(*context*, *node_id*, *trait*)
Check whether a Trait exists in the DB.

Parameters

- **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Trait(context).
- **node_id** The id of a node.
- **trait** A trait string.

Returns True if the trait exists otherwise False.

Raises NodeNotFound if the node no longer appears in the database.

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'node_id': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'trait': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property node_id

property trait

property updated_at

class ironic.objects.trait.TraitList(*args, **kwargs)

Bases: *ironic.objects.base.IronicObjectListBase*, *ironic.objects.base.IronicObject*

VERSION = '1.0'

classmethod create(*context*, *node_id*, *traits*)

Replace all existing traits with the specified list.

Parameters

- **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Trait(context).
- **node_id** The id of a node.
- **traits** List of Strings; traits to set.

Raises InvalidParameterValue if adding the trait would exceed the per-node traits limit.

Raises NodeNotFound if the node no longer appears in the database.

property created_at

dbapi = <oslo_db.api.DBAPI object>

classmethod destroy(*context*, *node_id*)

Delete all traits for the specified node.

Parameters

- **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Trait(context).
- **node_id** The id of a node.

Raises NodeNotFound if the node no longer appears in the database.

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'objects': List(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

classmethod get_by_node_id(*context*, *node_id*)
Return all traits for the specified node.

Parameters

- **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: Trait(context).
- **node_id** The id of a node.

Raises NodeNotFound if the node no longer appears in the database.

get_trait_names()
Return a list of names of the traits in this list.

property objects

property updated_at

ironic.objects.volume_connector module

class ironic.objects.volume_connector.**VolumeConnector**(*context=None*, ***kwargs*)
Bases: *ironic.objects.base.IronicObject*, oslo_versionedobjects.base.VersionedObjectDictCompat

VERSION = '1.0'

property connector_id

create(*context=None*)
Create a VolumeConnector record in the DB.

Parameters **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: VolumeConnector(context).

Raises VolumeConnectorTypeAndIdAlreadyExists if a volume connector already exists with the same type and connector_id

Raises VolumeConnectorAlreadyExists if a volume connector with the same UUID already exists

property created_at

dbapi = <oslo_db.api.DBAPI object>

destroy(*context=None*)
Delete the VolumeConnector from the DB.

Parameters **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: VolumeConnector(context).

Raises VolumeConnectorNotFound if the volume connector cannot be found

property extra

fields = {'connector_id': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'extra': FlexibleDict(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'id': Integer(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'node_id': Integer(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'type': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid': UUID(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

classmethod get(*context*, *ident*)

Find a volume connector based on its ID or UUID.

Parameters

- **context** security context
- **ident** the database primary key ID *or* the UUID of a volume connector

Returns a *VolumeConnector* object

Raises InvalidIdentity if ident is neither an integer ID nor a UUID

Raises VolumeConnectorNotFound if no volume connector exists with the specified ident

classmethod get_by_id(*context*, *db_id*)

Find a volume connector based on its integer ID.

Parameters

- **cls** the *VolumeConnector*
- **context** Security context.
- **db_id** The integer (database primary key) ID of a volume connector.

Returns A *VolumeConnector* object.

Raises VolumeConnectorNotFound if no volume connector exists with the specified ID.

classmethod get_by_uuid(*context*, *uuid*)

Find a volume connector based on its UUID.

Parameters

- **cls** the *VolumeConnector*
- **context** security context
- **uuid** the UUID of a volume connector

Returns a *VolumeConnector* object

Raises VolumeConnectorNotFound if no volume connector exists with the specified UUID

property id

classmethod list(*context*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)

Return a list of VolumeConnector objects.

Parameters

- **context** security context
- **limit** maximum number of resources to return in a single result
- **marker** pagination marker for large data sets
- **sort_key** column to sort results by
- **sort_dir** direction to sort. asc or desc.
- **project** The associated node project to search with.

Returns a list of *VolumeConnector* objects

Raises InvalidParameterValue if sort_key does not exist

classmethod list_by_node_id(*context*, *node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)

Return a list of VolumeConnector objects related to a given node ID.

Parameters

- **context** security context
- **node_id** the integer ID of the node
- **limit** maximum number of resources to return in a single result
- **marker** pagination marker for large data sets
- **sort_key** column to sort results by
- **sort_dir** direction to sort. asc or desc.
- **project** The associated node project to search with.

Returns a list of *VolumeConnector* objects

Returns a list of *VolumeConnector* objects

Raises InvalidParameterValue if sort_key does not exist

property node_id

refresh(*context=None*)

Load updates for this VolumeConnector.

Load a volume connector with the same UUID from the database and check for updated attributes. If there are any updates, they are applied from the loaded volume connector, column by column.

Parameters **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: VolumeConnector(context).

save(*context=None*)

Save updates to this VolumeConnector.

Updates will be made column by column based on the result of self.do_version_changes_for_db().

Parameters **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: VolumeConnector(context).

Raises VolumeConnectorNotFound if the volume connector cannot be found

Raises VolumeConnectorTypeAndIdAlreadyExists if another connector already exists with the same values for type and connector_id fields

Raises InvalidParameterValue when the UUID is being changed

property type

property updated_at

property uuid

class ironic.objects.volume_connector.**VolumeConnectorCRUDNotification**(*context=None*, ***kwargs*)

Bases: *ironic.objects.notification.NotificationBase*

Notification emitted at CRUD of a volume connector.

VERSION = '1.0'

property created_at

property event_type

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'event_type': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'level': NotificationLevel(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('debug', 'info', 'warning', 'error', 'critical')), 'payload': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'publisher': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property level

property payload

property publisher

property updated_at

class ironic.objects.volume_connector.**VolumeConnectorCRUDPayload**(*connector*, *node_uuid*)

Bases: *ironic.objects.notification.NotificationPayloadBase*

Payload schema for CRUD of a volume connector.

SCHEMA = {'connector_id': ('connector', 'connector_id'), 'created_at': ('connector', 'created_at'), 'extra': ('connector', 'extra'), 'type': ('connector', 'type'), 'updated_at': ('connector', 'updated_at'), 'uuid': ('connector', 'uuid')}

VERSION = '1.0'

property connector_id

property created_at

property extra

fields = {'connector_id': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'extra': FlexibleDict(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'node_uuid': UUID(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'type': String(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid': UUID(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False)}

property node_uuid

property type

property updated_at

property uuid

ironic.objects.volume_target module

class ironic.objects.volume_target.**VolumeTarget**(*context=None*, ***kwargs*)

Bases: *ironic.objects.base.IronicObject*, oslo_versionedobjects.base.VersionedObjectDictCompat

VERSION = '1.0'

property boot_index

create(*context=None*)

Create a VolumeTarget record in the DB.

Parameters **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: VolumeTarget(context).

Raises VolumeTargetBootIndexAlreadyExists if a volume target already exists with the same node ID and boot index

Raises VolumeTargetAlreadyExists if a volume target with the same UUID exists

property created_at

dbapi = <oslo_db.api.DBAPI object>

destroy(*context=None*)

Delete the VolumeTarget from the DB.

Parameters **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: VolumeTarget(context).

Raises VolumeTargetNotFound if the volume target cannot be found

property extra

```
fields = {'boot_index': Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'created_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'extra': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'id':
Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False),
'node_id': Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'properties': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'updated_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid':
UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'volume_id': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'volume_type': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}
```

classmethod get(*context*, *ident*)

Find a volume target based on its ID or UUID.

Parameters

- **context** security context
- **ident** the database primary key ID *or* the UUID of a volume target

Returns a *VolumeTarget* object

Raises InvalidIdentity if ident is neither an integer ID nor a UUID

Raises VolumeTargetNotFound if no volume target with this ident exists

classmethod get_by_id(*context*, *db_id*)

Find a volume target based on its database ID.

Parameters

- **cls** the *VolumeTarget*
- **context** security context
- **db_id** the database primary key (integer) ID of a volume target

Returns a *VolumeTarget* object

Raises VolumeTargetNotFound if no volume target with this ID exists

classmethod get_by_uuid(*context*, *uuid*)

Find a volume target based on its UUID.

Parameters

- **cls** the *VolumeTarget*
- **context** security context
- **uuid** the UUID of a volume target

Returns a *VolumeTarget* object

Raises VolumeTargetNotFound if no volume target with this UUID exists

property id

classmethod list(*context*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)

Return a list of VolumeTarget objects.

Parameters

- **context** security context
- **limit** maximum number of resources to return in a single result
- **marker** pagination marker for large data sets
- **sort_key** column to sort results by
- **sort_dir** direction to sort. asc or desc.
- **project** The associated node project to search with.

Returns a list of `VolumeConnector` objects

Returns a list of *VolumeTarget* objects

Raises InvalidParameterValue if sort_key does not exist

classmethod list_by_node_id(*context*, *node_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)

Return a list of VolumeTarget objects related to a given node ID.

Parameters

- **context** security context
- **node_id** the integer ID of the node
- **limit** maximum number of resources to return in a single result
- **marker** pagination marker for large data sets
- **sort_key** column to sort results by
- **sort_dir** direction to sort. asc or desc.
- **project** The associated node project to search with.

Returns a list of `VolumeConnector` objects

Returns a list of *VolumeTarget* objects

Raises InvalidParameterValue if sort_key does not exist

classmethod list_by_volume_id(*context*, *volume_id*, *limit=None*, *marker=None*, *sort_key=None*, *sort_dir=None*, *project=None*)

Return a list of VolumeTarget objects related to a given volume ID.

Parameters

- **context** security context
- **volume_id** the UUID of the volume

- **limit** maximum number of volume targets to return in a single result
- **marker** pagination marker for large data sets
- **sort_key** column to sort results by
- **sort_dir** direction to sort. asc or desc.

Returns a list of *VolumeTarget* objects

Raises InvalidParameterValue if sort_key does not exist

property node_id

property properties

refresh(*context=None*)

Loads updates for this VolumeTarget.

Load a volume target with the same UUID from the database and check for updated attributes. If there are any updates, they are applied from the loaded volume target, column by column.

Parameters **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: VolumeTarget(context).

Raises VolumeTargetNotFound if the volume target cannot be found

save(*context=None*)

Save updates to this VolumeTarget.

Updates will be made column by column based on the result of self.do_version_changes_for_db().

Parameters **context** security context. NOTE: This should only be used internally by the indirection_api. Unfortunately, RPC requires context as the first argument, even though we dont use it. A context should be set when instantiating the object, e.g.: VolumeTarget(context).

Raises InvalidParameterValue if the UUID is being changed

Raises VolumeTargetBootIndexAlreadyExists if a volume target already exists with the same node ID and boot index values

Raises VolumeTargetNotFound if the volume target cannot be found

property updated_at

property uuid

property volume_id

property volume_type

class ironic.objects.volume_target.**VolumeTargetCRUDNotification**(*context=None*, ***kwargs*)

Bases: *ironic.objects.notification.NotificationBase*

Notification emitted at CRUD of a volume target.

VERSION = '1.0'

property created_at

property event_type

fields = {'created_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'event_type': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'level': NotificationLevel(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False,valid_values=('debug', 'info', 'warning', 'error', 'critical')), 'payload': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'publisher': Object(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False), 'updated_at': DateTime(default=<class 'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}

property level

property payload

property publisher

property updated_at

class ironic.objects.volume_target.**VolumeTargetCRUDPayload**(*target*, *node_uuid*)

Bases: *ironic.objects.notification.NotificationPayloadBase*

SCHEMA = {'boot_index': ('target', 'boot_index'), 'created_at': ('target', 'created_at'), 'extra': ('target', 'extra'), 'properties': ('target', 'properties'), 'updated_at': ('target', 'updated_at'), 'uuid': ('target', 'uuid'), 'volume_id': ('target', 'volume_id'), 'volume_type': ('target', 'volume_type')}

VERSION = '1.0'

property boot_index

property created_at

property extra

```
fields = {'boot_index': Integer(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'created_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'extra': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'node_uuid': UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False),
'properties': FlexibleDict(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'updated_at': DateTime(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True), 'uuid':
UUID(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=False),
'volume_id': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True),
'volume_type': String(default=<class
'oslo_versionedobjects.fields.UnspecifiedDefault'>,nullable=True)}
```

```
property node_uuid
```

```
property properties
```

```
property updated_at
```

```
property uuid
```

```
property volume_id
```

```
property volume_type
```

Module contents

ironic.objects.**register_all**()

Submodules

ironic.version module

Module contents

8.1.6 Understanding the Ironics CI

Its important to understand the role of each job in the CI, how to add new jobs and how to debug failures that may arise. To facilitate that, we have created the documentation below.

Jobs description

The description of each jobs that runs in the CI when you submit a patch for *openstack/ironic* is visible in *Table. OpenStack Ironic CI jobs description*.

Table 1: Table. OpenStack Ironic CI jobs description

Job name	Description
ironic-tox-unit-with-driver-libs	Runs Ironic unit tests with the driver dependencies installed under Python3
ironic-tempest-functional-python3	Deploys Ironic in standalone mode and runs tempest functional tests that matches the regex *ironic_tempest_plugin.tests.api* under Python3
ironic-grenade	Deploys Ironic in a DevStack and runs upgrade for all enabled services.
ironic-standalone	Deploys Ironic in standalone mode and runs tempest tests that match the regex *ironic_standalone*.
ironic-standalone-redfish	Deploys Ironic in standalone mode and runs tempest tests that match the regex *ironic_standalone* using the redfish driver.
ironic-tempest-partition-bios-redfish-pxe	Deploys Ironic in DevStack, configured to use dib ramdisk partition image with *pxe* boot and *redfish* driver. Runs tempest tests that match the regex *ironic_tempest_plugin.tests.scenario*, also deploys 1 virtual baremetal.
ironic-tempest-partition-uefi-redfish-vmedia	Deploys Ironic in DevStack, configured to use dib ramdisk partition image with *vmedia* boot and *redfish* driver. Runs tempest tests that match the regex *ironic_tempest_plugin.tests.scenario*, also deploys 1 virtual baremetal.
ironic-tempest-wholedisk-bios-snmp-pxe	Deploys Ironic in DevStack, configured to use a pre-built dib ramdisk wholedisk image that is downloaded from a Swift temporary url, *pxe* boot and *snmp* driver. Runs tempest tests that match the regex *ironic_tempest_plugin.tests.scenario* and deploys 1 virtual baremetal.
ironic-tempest-partition-bios-ipmi-pxe	Deploys Ironic in DevStack, configured to use dib ramdisk, a partition image, *pxe* boot in legacy mode and *ipmi* hardware type. Runs tempest tests that match the regex *ironic_tempest_plugin.tests.scenario* and deploy 1 virtual baremetal.
ironic-tempest-partition-uefi-ipmi-pxe	Deploys Ironic in DevStack, configured to use dib ramdisk, a partition image, *pxe* boot in UEFI mode and *ipmi* hardware type. Runs tempest tests that match the regex *ironic_tempest_plugin.tests.scenario*, also deploys 1 virtual baremetal.
ironic-tempest-ipa-wholedisk-direct-tinyipa-multinode	Deploys Ironic in a multinode DevStack, configured to use a pre-build tinyipa ramdisk wholedisk image that is downloaded from a Swift temporary url, *pxe* boot and *ipmi* driver. Runs tempest tests that match the regex *(ironic_tempest_plugin.tests.scenario|test_schedule_to_all_nod* and deploys 7 virtual baremetal.
ironic-tempest-bios-ipmi-direct-tinyipa	Deploys Ironic in DevStack, configured to

Adding a new Job

Are you familiar with Zuul?

Before start trying to figure out how Zuul works, take some time and read about Zuul Config and the Zuul Best Practices.

Where can I find the existing jobs?

The jobs for the Ironic project are defined under the zuul.d folder in the root directory, that contains three files, whose function is described below.

- ironic-jobs.yaml: Contains the configuration of each Ironic Job converted to Zuul v3.
- legacy-ironic-jobs.yaml: Contains the configuration of each Ironic Job that havent been converted to Zuul v3 yet.
- project.yaml: Contains the jobs that will run during check and gate phase.

Create a new Job

Identify among the existing jobs the one that most closely resembles the scenario you want to test, the existing job will be used as *parent* in your job definition. Now you will only need to either overwrite or add variables to your job definition under the *vars* section to represent the desired scenario.

The code block below shows the minimal structure of a new job definition that you need to add to ironic-jobs.yaml.

```
- job:
    name: <name of the new job>
    description: <what your job does>
    parent: <Job that already exists>
    vars:
      <var1>: <new value>
```

After having the definition of your new job you just need to add the job name to the project.yaml under *check* and *gate*. Only jobs that are voting should be in the *gate* section.

```
- project:
    check:
      jobs:
        - <name of the new job>
    gate:
      queue: ironic
      jobs:
        - <name of the new job>
```

Debugging CI failures

If you see *FAILURE* in one or more jobs for your patch please dont panic. This guide may help you to find the initial reason for the failure. When clicking in the failed job you will be redirect to the Zuul web page that contains all the information about the job build.

Zuul Web Page

The page has three tabs: *Summary*, *Logs* and *Console*.

- Summary: Contains overall information about the build of the job, if the job build failed it will contain a general output of the failure.
- Logs: Contains all configurations and log files about all services that were used in the job. This will give you an overall idea of the failures and you can identify services that may be involved. The *job-output* file can give an overall idea of the failures and what services may be involved.
- Console: Contains all the playbooks that were executed, by clicking in the arrow before each playbook name you can find the roles and commands that were executed.

8.1.7 Our policy for stable branches

Stable branches that are on Extended Maintenance and havent received backports in a while, can be tagged as `Unmaintained`, after discussions within the ironic community. If such a decision is taken, an email will be sent to the OpenStack mailing list.

What does `Unmaintained` mean? The branch still exists, but the ironic upstream community will not actively backport patches from maintained branches. Fixes can still be merged, though, if pushed into review by operators or other downstream developers. It also means that branchless projects (e.g.: ironic-tempest-plugin), may not have configurations that are compatible with those branches.

As of 09 March 2020, the list of `Unmaintained` branches includes:

- Ocata (Last commit - Jun 28, 2019)
- Pike (Last commit - Oct 2, 2019)

CHAPTER

NINE

RELEASE NOTES

Release Notes

PYTHON MODULE INDEX

a

c

d

i

o

v

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X