Ironic Python Agent Documentation

Release 6.4.5.dev2

OpenStack Foundation

Jan 27, 2023

CONTENTS

CHAPTER ONE

OVERVIEW

Ironic Python Agent (often abbreviated as IPA) is an agent for controlling and deploying Ironic controlled baremetal nodes. Typically run in a ramdisk, the agent exposes a REST API for provisioning servers.

CHAPTER

TWO

CONTENTS

2.1 Installing Ironic Python Agent

2.1.1 Image Builders

Unlike most other python software, you must build or download an IPA ramdisk image before use. This is because its not installed in an operating system, but instead is run from within a ramdisk.

Two kinds of images are published on every commit from every branch of IPA:

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

If you need to build your own image, use the tools from the ironic-python-agent-builder project.

2.1.2 IPA Flags

You can pass a variety of flags to IPA on start up to change its behavior.

- `--standalone`: This disables the initial lookup and heartbeats to Ironic. Lookup sends some information to Ironic in order to determine Ironics node UUID for the node. Heartbeat sends periodic pings to Ironic to tell Ironic the node is still running. These heartbeats also trigger parts of the deploy and cleaning cycles. This flag is useful for debugging IPA without an Ironic installation.
- `--debug`: Enables debug logging.

2.1.3 IPA and TLS

Client Configuration

During its operation IPA makes HTTP requests to a number of other services, currently including

- ironic for lookup/heartbeats
- ironic-inspector to publish results of introspection
- HTTP image storage to fetch the user image to be written to the nodes disk (Object storage service or other service storing user images when ironic is running in a standalone mode)

When these services are configured to require TLS-encrypted connections, IPA can be configured to either properly use such secure connections or ignore verifying such TLS connections.

Configuration mostly happens in the IPA config file (default is `/etc/ironic_python_agent/ironic_python_agent.conf`, can also be any file placed in `/etc/ironic-python-agent.d`) or command line arguments passed to `ironic-python-agent`, and it is possible to provide some options via kernel command line arguments instead.

Available options in the `[DEFAULT]` config file section are:

insecure Whether to verify server TLS certificates. When not specified explicitly, defaults to the value of `ipa-insecure` kernel command line argument (converted to boolean). The default for this kernel command line argument is taken to be `False`. Overriding it to `True` by adding `ipa-insecure=1` to the value of `[pxe]pxe_append_params` in ironic configuration file will allow running the same IPA-based deploy ramdisk in a CI-like environment when services are using secure HTTPS endpoints with self-signed certificates without adding a custom CA file to the deploy ramdisk (see below).

cafile Path to the PEM encoded Certificate Authority file. When not specified, available system-wide list of CAs will be used to verify server certificates. Thus in order to use IPA with HTTPS endpoints of other services in a secure fashion (with `insecure` option being `False`, see above), operators should either ensure that certificates of those services are verifiable by root CAs present in the deploy ramdisk, or add a custom CA file to the ramdisk and set this IPA option to point to this file at ramdisk build time.

certfile Path to PEM encoded client certificate cert file. This option must be used when services are configured to require client certificates on SSL-secured connections. This cert file must be added to the deploy ramdisk and path to it specified for IPA via this option at ramdisk build time. This option has an effect only when the `keyfile` option is also set.

keyfile Path to PEM encoded client certificate key file. This option must be used when services are configured to require client certificates on SSL-secured connections. This key file must be added to the deploy ramdisk and path to it specified for IPA via this option at ramdisk build time. This option has an effect only when the `certfile` option is also set.

Currently a single set of cafile/certfile/keyfile options is used for all HTTP requests to the other services.

Server Configuration

Starting with the Victoria release, the API provided by ironic-python-agent can also be secured via TLS. There are two options to do that:

Automatic TLS This option is enabled by default if no other options are enabled. If ironic supports API version 1.68, a new self-signed TLS certificate will be generated in runtime and sent to ironic on heartbeat.

No special configuration is required on the ironic side.

Manual TLS If you need to provide your own TLS certificate, you can configure it when building an image. Set the following options in the ironic-python-agent configuration file:

```
[DEFAULT]
listen_tls = True
advertise_protocol = https
# Disable automatic TLS.
enable_auto_tls = False
```

(continues on next page)

(continued from previous page)

```
[ssl]
# Certificate and private key file paths (on the ramdisk).
cert_file = /path/to/certificate
# The private key must not be password-protected!
key_file = /path/to/private/key
# Optionally, authenticate connecting clients (i.e. ironic␣
↪conductors).
#ca_file = /path/to/ca
```

If using DIB to build the ramdisk, use the ironic-python-agent-tls element to automate these steps.

On the ironic side you have two options:

- If the certificate can pass host validation, i.e. contains the correct host name or IP address of the agent, add its path to each node with:

  ```
  baremetal node set <node> --driver-info agent_verify_ca=/path/to/
  ↪ca/or/certificate
  ```

- Usually, the IP address of the agent is not known in advance, so you need to disable host validation instead:

  ```
  baremetal node set <node> --driver-info agent_verify_ca=False
  ```

2.1.4 Hardware Managers

Hardware managers are how IPA supports multiple different hardware platforms in the same agent. Any action performed on hardware can be overridden by deploying your own hardware manager.

Custom hardware managers allow you to include hardware-specific tools, files and cleaning steps in the Ironic Python Agent. For example, you could include a BIOS flashing utility and BIOS file in a custom ramdisk. Your custom hardware manager could expose a cleaning step that calls the flashing utility and flashes the packaged BIOS version (or even download it from a tested web server).

Operators wishing to build their own hardware managers should reference the documentation available at *Hardware Managers*.

2.2 Ironic Python Agent Administration

2.2.1 How it works

Integration with Ironic

For information on how to install and configure Ironic drivers, including drivers for IPA, see the Ironic drivers documentation.

Lookup

On startup, the agent performs a lookup in Ironic to determine its node UUID by sending a hardware profile to the Ironic lookup endpoint: /v1/lookup.

Heartbeat

After successfully looking up its node, the agent heartbeats via /v1/heartbeat/{node_ident} every N seconds, where N is the Ironic conductors `agent.heartbeat_timeout` value multiplied by a number between .3 and .6.

For example, if your conductors ironic.conf contains:

```
[agent]
heartbeat_timeout = 60
```

IPA will heartbeat between every 20 and 36 seconds. This is to ensure jitter for any agents reconnecting after a network or API disruption.

After the agent heartbeats, the conductor performs any actions needed against the node, including querying status of an already run command. For example, initiating in-band cleaning tasks or deploying an image to the node.

Inspection

IPA can conduct hardware inspection on start up and post data to the Ironic Inspector via the /v1/continue endpoint.

Edit your default PXE/iPXE configuration or IPA options baked in the image, and set `ipa-inspection-callback-url` to the full endpoint of Ironic Inspector, for example:

```
ipa-inspection-callback-url=http://IP:5050/v1/continue
```

Make sure your DHCP environment is set to boot IPA by default.

For the cases where the infrastructure operator and cloud user are the same, an additional tool exists that can be installed alongside the agent inside a running instance. This is the `ironic-collect-introspection-data` command which allows for a node in `ACTIVE` state to publish updated introspection data to ironic-inspector. This ability requires ironic-inspector to be configured with `[processing]permit_active_introspection` set to `True`. For example:

```
ironic-collect-introspection-data --inspection_callback_url http://IP:5050/
↪v1/continue
```

Alternatively, this command may also be used with multicast DNS functionality to identify the Ironic Inspector service endpoint. For example:

```
ironic-collect-introspection-data --inspection_callback_url mdns
```

An additional daemon mode may be useful for some operators who wish to receive regular updates, in the form of the `[DEFAULT]introspection_daemon` boolean configuration option. For example:

```
ironic-collect-introspection-data --inspection_callback_url mdns --
↪introspection_daemon
```

The above command will attempt to connect to introspection and will then enter a loop to publish every 300 seconds. This can be tuned with the `[DEFAULT]introspection_daemon_post_interval` configuration option.

Inspection Data

As part of the inspection process, data is collected on the machine and sent back to Ironic Inspector for storage. It can be accessed via the introspection data API.

The exact format of the data depends on the enabled *collectors*, which can be configured using the `ipa-inspection-collectors` kernel parameter. Each collector appends information to the resulting JSON object. The in-tree collectors are:

`default` The default enabled collectors. Collects the following keys:

- `inventory` - *Hardware Inventory*.
- `root_disk` - The default root device for this machine, which will be used for deployment if root device hints are not provided.
- `configuration` - Inspection configuration, an object with two keys:
  - `collectors` - List of enabled collectors.
  - `managers` - List of enabled *Hardware Managers*: items with keys `name` and `version`.
- `boot_interface` - Deprecated, use the `inventory.boot.pxe_interface` field.

`logs` Collect system logs. To yield useful results it must always go last in the list of collectors. Provides one key:

- `logs` - base64 encoded tarball with various logs.

`pci-devices` Collects the list of PCI devices. Provides one key:

- `pci_devices` - list of objects with keys `vendor_id` and `product_id`.

`extra-hardware` Collects a vast list of facts about the systems, using the hardware library, which is a required dependency for this collector. Adds one key:

- `data` - raw data from the `hardware-collect` utility. Is a list of lists with 4 items each. It is recommended to use this collector together with the `extra_hardware` processing hook on the Ironic Inspector side to convert it to a nested dictionary in the `extra` key.

  If `ipa-inspection-benchmarks` is set, the corresponding benchmarks are executed and their result is also provided.

`dmi-decode` Collects information from `dmidecode`. Provides one key:

- `dmi` DMI information in three keys: `bios`, `cpu` and `memory`.

`numa-topology` Collects NUMA topology information. Provides one key:

- `numa_topology` with three nested keys:
  - `ram` - list of objects with keys `numa_node` (node ID) and `size_kb`.

  - `cpus` - list of objects with keys `cpu` (CPU ID), `numa_node` (node ID) and `thread_siblings` (list of sibling threads).
  - `nics` - list of objects with keys `name` (NIC name) and `numa_node` (node ID).

Hardware Inventory

IPA collects various hardware information using its *Hardware Managers*, and sends it to Ironic on lookup and to Ironic Inspector on *Inspection*.

The exact format of the inventory depends on the hardware manager used. Here is the basic format expected to be provided by all hardware managers. The inventory is a dictionary (JSON object), containing at least the following fields:

`cpu` CPU information: `model_name`, `frequency`, `count`, `architecture` and `flags`.

`memory` RAM information: `total` (total size in bytes), `physical_mb` (physically installed memory size in MiB, optional).

> **Note:** The difference is that the latter includes the memory region reserved by the kernel and is always slightly bigger. It also matches what the Nova flavor would contain for this node and thus is used by the inspection process instead of `total`.

`bmc_address` IPv4 address of the nodes BMC (aka IPMI v4 address), optional.

`bmc_v6address` IPv6 address of the nodes BMC (aka IPMI v6 address), optional.

`disks` list of disk block devices with fields: `name`, `model`, `size` (in bytes), `rotational` (boolean), `wwn`, `serial`, `vendor`, `wwn_with_extension`, `wwn_vendor_extension`, `hctl` and `by_path` (the full disk path, in the form `/dev/disk/by-path/<rest-of-path>`).

`interfaces` list of network interfaces with fields: `name`, `mac_address`, `ipv4_address`, `lldp`, `vendor`, `product`, and optionally `` biosdevname`` (BIOS given NIC name). If configuration option ``collect_lldp `` is set to True the `lldp` field will be populated by a list of type-length-value(TLV) fields retrieved using the Link Layer Discovery Protocol (LLDP).

`system_vendor` system vendor information from SMBIOS as reported by `dmidecode`: `product_name`, `serial_number` and `manufacturer`.

`boot` boot information with fields: `current_boot_mode` (boot mode used for the current boot - BIOS or UEFI) and `pxe_interface` (interface used for PXE booting, if any).

`hostname` hostname for the system

> **Note:** This is most likely to be set by the DHCP server. Could be localhost if the DHCP server does not set it.

2.2.2 Built-in hardware managers

GenericHardwareManager

This is the default hardware manager for ironic-python-agent. It provides support for *Hardware Inventory* and the default deploy and clean steps.

Deploy steps

`deploy.write_image(node, ports, image_info, configdrive=None)` A deploy step backing the `write_image` deploy step of the direct deploy interface. Should not be used explicitly, but can be overridden to provide a custom way of writing an image.

`deploy.erase_devices_metadata(node, ports)` Erases partition tables from all recognized disk devices. Can be used with software RAID since it requires empty holder disks.

`raid.apply_configuration(node, ports, raid_config, delete_existing=True)` Apply a software RAID configuration. It belongs to the `raid` interface and must be used through the ironic RAID feature.

Clean steps

`deploy.erase_devices` Securely erases all information from all recognized disk devices. Relatively fast when secure ATA erase is available, otherwise can take hours, especially on a virtual environment. Enabled by default.

`deploy.erase_devices_metadata` Erases partition tables from all recognized disk devices. Can be used as an alternative to the much longer `erase_devices` step.

`raid.create_configuration` Create a RAID configuration. This step belongs to the `raid` interface and must be used through the ironic RAID feature.

`raid.delete_configuration` Delete the RAID configuration. This step belongs to the `raid` interface and must be used through the ironic RAID feature.

2.2.3 Rescue mode

Overview

Rescue mode is a feature that can be used to boot a ramdisk for a tenant in case the machine is otherwise inaccessible. For example, if theres a disk failure that prevents access to another operating system, rescue mode can be used to diagnose and fix the problem.

Support in ironic-python-agent images

Rescue is initiated when ironic-conductor sends the `finalize_rescue` command to ironic-python-agent. A user *rescue* is created with a password provided as an argument to this command. DHCP is then configured to facilitate network connectivity, thus enabling a user to login to the machine in rescue mode.

> **Warning:** Rescue mode exposes the contents of the ramdisk to the tenant. Ensure that any rescue image you build does not contain secrets (e.g. sensitive clean steps, proprietary firmware blobs).

The below has information about supported images that may be built to use rescue mode.

DIB

The DIB image supports rescue mode when used with DHCP tenant networks.

After the `finalize_rescue` command completes, DHCP will be configured on all network interfaces, and a *rescue* user will be created with the specified `rescue_password`.

TinyIPA

The TinyIPA image supports rescue mode when used with DHCP tenant networks. No special action is required to build a TinyIPA image with this support.

After the `finalize_rescue` command completes, DHCP will be configured on all network interfaces, and a *rescue* user will be created with the specified `rescue_password`.

2.2.4 Troubleshooting Ironic-Python-Agent (IPA)

This document contains basic trouble shooting information for IPA.

Gaining Access to IPA on a node

In order to access a running IPA instance a user must be added or enabled on the image. Below we will cover several ways to do this.

Access via ssh

ironic-python-agent-builder

SSH access can be added to DIB built IPA images with the dynamic-login[0] or the devuser element[1]

The dynamic-login element allows the operator to inject a SSH key when the image boots. Kernel command line parameters are used to do this.

[0] *Dynamic-login DIB element*: https://github.com/openstack/diskimage-builder/tree/master/diskimage_builder/elements/dynamic-login

[1] *DevUser DIB element*: https://github.com/openstack/diskimage-builder/tree/master/diskimage_builder/elements/devuser

dynamic-login element example:

- Add `sshkey="ssh-rsa BBA1..."` to pxe_append_params setting in the `ironic.conf` file
- Restart the ironic-conductor with the command `service ironic-conductor restart`

Install `ironic-python-agent-builder` following the guide[2]

devuser element example:

```
export DIB_DEV_USER_USERNAME=username
export DIB_DEV_USER_PWDLESS_SUDO=yes
export DIB_DEV_USER_AUTHORIZED_KEYS=$HOME/.ssh/id_rsa.pub
ironic-python-agent-builder -o /path/to/custom-ipa -e devuser debian
```

tinyipa

If you want to enable SSH access to the image, set `AUTHORIZE_SSH` variable in your shell to `true` before building the tinyipa image:

```
export AUTHORIZE_SSH=true
```

By default it will use default public RSA (or, if not available, DSA) key of the user running the build (`~/.ssh/id_{rsa,dsa}.pub`).

To provide other public SSH key, export full path to it in your shell before building tinyipa as follows:

```
export SSH_PUBLIC_KEY=/path/to/other/ssh/public/key
```

The user to use for access is default Tiny Core Linux user `tc`. This user has no password and has password-less `sudo` permissions. Installed SSH server is configured to disable Password authentication.

Access via console

If you need to use console access, passwords must be enabled there are a couple ways to enable this depending on how the IPA image was created:

ironic-python-agent-builder

Users wishing to use password access can be add the dynamic-login[0] or the devuser element[1]

The dynamic-login element allows the operator to change the root password dynamically when the image boots. Kernel command line parameters are used to do this.

dynamic-login element example:

```
Generate a ENCRYPTED_PASSWORD with the openssl passwd -1 command
Add rootpwd="$ENCRYPTED_PASSWORD" value on the pxe_append_params setting␣
↪in /etc/ironic/ironic.conf
Restart the ironic-conductor with the command service ironic-conductor␣
↪restart
```

[2] *ironic-python-agent-builder*: https://docs.openstack.org/ironic-python-agent-builder/latest/install/index.html

Users can also be added to DIB built IPA images with the devuser element[1]

Install `ironic-python-agent-builder` following the guide[2]

Example:

```
export DIB_DEV_USER_USERNAME=username
export DIB_DEV_USER_PWDLESS_SUDO=yes
export DIB_DEV_USER_PASSWORD=PASSWORD
ironic-python-agent-builder -o /path/to/custom-ipa -e devuser debian
```

tinyipa

The image built with scripts provided in `tinyipa` folder of Ironic Python Agent Builder repository by default auto-logins the default Tiny Core Linux user `tc` to the console. This user has no password and has password-less `sudo` permissions.

How to pause the IPA for debugging

When debugging issues with the IPA, in particular with cleaning, it may be necessary to log in to the RAM disk before the IPA actually starts (and delay the launch of the IPA). One easy way to do this is to set `maintenance` on the node and then trigger cleaning. Ironic will boot the node into the RAM disk, but the IPA will stall until the maintenance state is removed. This opens a time window to log into the node.

Another way to do this is to add simple cleaning steps in a custom hardware manager which sleep until a certain condition is met, e.g. until a given file exists. Having multiple of these barrier steps allows to go through the cleaning steps and have a break point in between them.

Set IPA to debug logging

Debug logging can be enabled a several different ways. The easiest way is to add `ipa-debug=1` to the kernel command line. To do this:

- Append `ipa-debug=1` to the pxe_append_params setting in the `ironic.conf` file
- Restart the ironic-conductor with the command `service ironic-conductor restart`

If the system is running and uses systemd then editing the services file will be required.

- `systemctl edit ironic-python-agent.service`
- Append `--debug` to end of the ExecStart command
- Restart IPA. See the *Manually restart IPA* section below.

Where can I find the IPA logs

Retrieving the IPA logs will differ depending on which base image was used.

- Operating system that do not use `systemd` (ie Ubuntu 14.04)
  - logs will be found in the /var/log/ folder.
- Operating system that do use `systemd` (ie Fedora, CentOS, RHEL)
  - logs may be viewed with `sudo journalctl -u ironic-python-agent`
  - if using a diskimage-builder ramdisk, it may be configured to output all contents of the journal, including ironic-python-agent logs, by enabling the journal-to-console element.

Manually restart IPA

In some cases it is helpful to enable debug mode on a running node. If the system does not use systemd then IPA can be restarted directly:

```
sudo /usr/local/bin/ironic-python-agent [--debug]
```

If the system uses systemd then systemctl can be used to restart the service:

```
sudo systemctl restart ironic-python-agent.service
```

References

2.3 Contributing to Ironic Python Agent

Ironic Python Agent is an agent for controlling and deploying Ironic controlled baremetal nodes. Typically run in a ramdisk, the agent exposes a REST API for provisioning servers.

Throughout the remainder of the document, Ironic Python Agent will be abbreviated to IPA.

2.3.1 Hardware Managers

Hardware managers are how IPA supports multiple different hardware platforms in the same agent. Any action performed on hardware can be overridden by deploying your own hardware manager.

IPA ships with *GenericHardwareManager*, which implements basic cleaning and deployment methods compatible with most hardware.

How are methods executed on HardwareManagers?

Methods that modify hardware are dispatched to each hardware manager in priority order. When a method is dispatched, if a hardware manager does not have a method by that name or raises *IncompatibleHardwareMethodError*, IPA continues on to the next hardware manager. Any hardware manager that returns a result from the method call is considered a success and its return value passed on to whatever dispatched the method. If the method is unable to run successfully on any hardware managers, *HardwareManagerMethodNotFound* is raised.

Why build a custom HardwareManager?

Custom hardware managers allow you to include hardware-specific tools, files and cleaning steps in the Ironic Python Agent. For example, you could include a BIOS flashing utility and BIOS file in a custom ramdisk. Your custom hardware manager could expose a cleaning step that calls the flashing utility and flashes the packaged BIOS version (or even download it from a tested web server).

How can I build a custom HardwareManager?

In general, custom HardwareManagers should subclass hardware.HardwareManager. Subclassing hardware.GenericHardwareManager should only be considered if the aim is to raise the priority of all methods of the GenericHardwareManager. The only required method is evaluate_hardware_support(), which should return one of the enums in hardware.HardwareSupport. Hardware support determines which hardware manager is executed first for a given function (see: *How are methods executed on HardwareManagers?* for more info). Common methods you may want to implement are `list_hardware_info()`, to add additional hardware the GenericHardwareManager is unable to identify and `erase_devices()`, to erase devices in ways other than ATA secure erase or shredding.

Some reusable functions are provided by ironic-lib, its IPA is relatively stable.

The examples directory has two example hardware managers that can be copied and adapter for your use case.

Custom HardwareManagers and Cleaning

One of the reasons to build a custom hardware manager is to expose extra steps in Ironic Cleaning. A node will perform a set of cleaning steps any time the node is deleted by a tenant or moved from `manageable` state to `available` state. Ironic will query IPA for a list of clean steps that should be executed on the node. IPA will dispatch a call to *get_clean_steps()* on all available hardware managers and then return the combined list to Ironic.

To expose extra clean steps, the custom hardware manager should have a function named *get_clean_steps()* which returns a list of dictionaries. The dictionaries should be in the form:

```
def get_clean_steps(self, node, ports):
    return [
        {
            # A function on the custom hardware manager
            'step': 'upgrade_firmware',
            # An integer priority. Largest priorities are executed first
            'priority': 10,
            # Should always be the deploy interface
```

(continues on next page)

(continued from previous page)

```
            'interface': 'deploy',
            # Request the node to be rebooted out of band by Ironic when
            # the step completes successfully
            'reboot_requested': False
        }
    ]
```

Then, you should create functions which match each of the *step* keys in the clean steps you return. The functions will take two parameters: *node*, a dictionary representation of the Ironic node, and *ports*, a list of dictionary representations of the Ironic ports attached to *node*.

When a clean step is executed in IPA, the *step* key will be sent to the hardware managers in hardware support order, using *hardware.dispatch_to_managers()*. For each hardware manager, if the manager has a function matching the *step* key, it will be executed. If the function returns a value (including None), that value is returned to Ironic and no further managers are called. If the function raises *IncompatibleHardwareMethodError*, the next manager will be called. If the function raises any other exception, the command will be considered failed, the command results error message will be set to the exceptions error message, and no further managers will be called. An example step:

```
def upgrade_firmware(self, node, ports):
    if self._device_exists():
        # Do the upgrade
        return 'upgraded firmware'
    else:
        raise errors.IncompatibleHardwareMethodError()
```

Note: If two managers return steps with the same *step* key, the priority will be set to whichever manager has a higher hardware support level and then use the higher priority in the case of a tie.

Custom HardwareManagers and Deploying

Starting with the Victoria release cycle, deployment can be customized similarly to cleaning. A hardware manager can define *deploy steps* that may be run during deployment by exposing a `get_deploy_steps` call.

There are two kinds of deploy steps:

1. Steps that need to be run automatically must have a non-zero priority and cannot take required arguments. For example:

```
def get_deploy_steps(self, node, ports):
    return [
        {
            # A function on the custom hardware manager
            'step': 'upgrade_firmware',
            # An integer priority. Largest priorities are executed␣
↪first
            'priority': 10,
            # Should always be the deploy interface
            'interface': 'deploy',
        }
```

(continues on next page)

(continued from previous page)

```
    ]

# A deploy steps looks the same as a clean step.
def upgrade_firmware(self, node, ports):
    if self._device_exists():
        # Do the upgrade
        return 'upgraded firmware'
    else:
        raise errors.IncompatibleHardwareMethodError()
```

Priority should be picked based on when exactly in the process the step will run. See agent step priorities for guidance.

2. Steps that will be requested via deploy templates should have a priority of 0 and may take both required and optional arguments that will be provided via the deploy templates. For example:

```
def get_deploy_steps(self, node, ports):
    return [
        {
            # A function on the custom hardware manager
            'step': 'write_a_file',
            # Steps with priority 0 don't run by default.
            'priority': 0,
            # Should be the deploy interface, unless there is driver-
↪side
            # support for another interface (as it is for RAID).
            'interface': 'deploy',
            # Arguments that can be required or optional.
            'argsinfo': {
                'path': {
                    'description': 'Path to file',
                    'required': True,
                },
                'content': {
                    'description': 'Content of the file',
                    'required': True,
                },
                'mode': {
                    'description': 'Mode of the file, defaults to 0644
↪',
                    'required': False,
                },
            }
        }
    ]

def write_a_file(self, node, ports, path, contents, mode=0o644):
    pass  # Mount the disk, write a file.
```

Versioning

Each hardware manager has a name and a version. This version is used during cleaning to ensure the same version of the agent is used to on a node through the entire process. If the version changes, cleaning is restarted from the beginning to ensure consistent cleaning operations and to make updating the agent in production simpler.

You can set the version of your hardware manager by creating a class variable named HARDWARE_MANAGER_VERSION, which should be a string. The default value is 1.0. You should change this version string any time you update your hardware manager. You can also change the name your hardware manager presents by creating a class variable called HARDWARE_MANAGER_NAME, which is a string. The name defaults to the class name. Currently IPA only compares version as a string; any version change whatsoever will induce cleaning to restart.

Priority

A hardware manager has a single overall priority, which should be based on how well it supports a given piece of hardware. At load time, IPA executes *evaluate_hardware_support()* on each hardware manager. This method should return an int representing hardware manager priority, based on what it detects about the platform its running on. Suggested values are included in the *HardwareSupport* class. Returning a value of 0 aka *HardwareSupport.NONE*, will prevent the hardware manager from being used. IPA will never ship a hardware manager with a priority higher than 3, aka *HardwareSupport.SERVICE_PROVIDER*.

2.3.2 Emitting metrics from Ironic-Python-Agent (IPA)

This document describes how to emit metrics from IPA, including timers and counters in code to directly emitting hardware metrics from a custom HardwareManager.

Overview

IPA uses the metrics implementation from ironic-lib, with a few caveats due to the dynamic configuration done at lookup time. You cannot cache the metrics instance as the MetricsLogger returned will change after lookup if configs different than the default setting have been used. This also means that the method decorator supported by ironic-lib cannot be used in IPA.

Using a context manager

Using the context manager is the recommended way for sending metrics that time or count sections of code. However, given that you cannot cache the MetricsLogger, you have to explicitly call get_metrics_logger() from ironic-lib every time. For example:

```
from ironic_lib import metrics_utils

def my_method():
    with metrics_utils.get_metrics_logger(__name__).timer('my_method'):
        return _do_work()
```

As a note, these metric collectors do work for custom HardwareManagers as well. However, you may want to metric the portions of a method that determine compatibility separate from portions of a method that actually do work, in order to assure the metrics are relevant and useful on all hardware.

Explicitly sending metrics

A feature that may be particularly helpful for deployers writing custom HardwareManagers is the ability to explicitly send metrics. For instance, you could add a cleaning step which would retrieve metrics about a device and ship them using the provided metrics library. For example:

```
from ironic_lib import metrics_utils

def my_cleaning_step():
    for name, value in _get_smart_data():
        metrics_utils.get_metrics_logger(__name__).send_gauge(name, value)
```

References

For more information, please read the source of the metrics module in ironic-lib.

2.3.3 Rescue Mode

Ironic supports putting nodes in rescue mode using hardware types that support rescue interfaces. A rescue operation can be used to boot nodes into a rescue ramdisk so that the `rescue` user can access the node. This provides the ability to access the node when normal access is not possible. For example, if there is a need to perform manual password reset or data recovery in the event of some failure, a rescue operation can be used. IPA rescue extension exposes a command `finalize_rescue` (that is used by Ironic) to set the password for the `rescue` user when the rescue ramdisk is booted.

finalize_rescue command

The rescue extension exposes the command `finalize_rescue`; when invoked, it triggers rescue mode:

```
POST /v1/commands

{"name": "rescue.finalize_rescue",
 "params": {
    "rescue_password": "p455w0rd"}
}
```

`rescue_password` is a required parameter for this command.

Upon success, it returns following data in response:

```
{"command_name": "finalize_rescue",
 "command_params": {
    "rescue_password": "p455w0rd"},
 "command_status": "SUCCEEDED"
 "command_result": null
```

(continues on next page)

(continued from previous page)

```
  "command_error": null
}
```

If successful, this synchronous command will:

1. Write the salted and crypted `rescue_password` to `/etc/ipa-rescue-config/ipa-rescue-password` in the chroot or filesystem that ironic-python-agent is running in.
2. Stop the ironic-python-agent process after completing these actions and returning the response to the API request.

2.3.4 Generated Developer Documentation

- modindex

ironic_python_agent

ironic_python_agent package

Subpackages

ironic_python_agent.api package

Submodules

ironic_python_agent.api.app module

class `ironic_python_agent.api.app.`**`Application`**(*agent*, *conf*)

Bases: `object`

`api_get_command`(*request*, *cmd*)

`api_list_commands`(*request*)

`api_root`(*request*)

`api_run_command`(*request*)

`api_status`(*request*)

`api_v1`(*request*)

`handle_exception`(*environ*, *exc*)

Handle an exception during request processing.

`start`(*tls_cert_file=None*, *tls_key_file=None*)

Start the API service in the background.

`stop`()

Stop the API service.

class ironic_python_agent.api.app.**Request**(*environ*, *populate_request=True*, *shallow=False*)

Bases: `werkzeug.wrappers.request.Request`, `werkzeug.wrappers.json.JSONMixin`

Custom request class with JSON support.

ironic_python_agent.api.app.**format_exception**(*value*)

ironic_python_agent.api.app.**jsonify**(*value*, *status=200*)

Convert value to a JSON response using the custom encoder.

ironic_python_agent.api.app.**make_link**(*url*, *rel_name*, *resource=''*, *resource_args=''*, *bookmark=False*, *type_=None*)

ironic_python_agent.api.app.**version**(*url*)

Module contents

ironic_python_agent.cmd package

Submodules

ironic_python_agent.cmd.agent module

ironic_python_agent.cmd.agent.**run**()

Entrypoint for IronicPythonAgent.

ironic_python_agent.cmd.inspect module

ironic_python_agent.cmd.inspect.**run**()

Entrypoint for IronicPythonAgent.

Module contents

ironic_python_agent.extensions package

Submodules

ironic_python_agent.extensions.base module

class ironic_python_agent.extensions.base.**AgentCommandStatus**

Bases: `object`

Mapping of agent command statuses.

FAILED = 'FAILED'

RUNNING = 'RUNNING'

SUCCEEDED = 'SUCCEEDED'

VERSION_MISMATCH = 'CLEAN_VERSION_MISMATCH'

class ironic_python_agent.extensions.base.**AsyncCommandResult**(*command_name*, *command_params*, *execute_method*, *agent=None*)

Bases: *ironic_python_agent.extensions.base.BaseCommandResult*

A command that executes asynchronously in the background.

is_done()

Checks to see if command is still RUNNING.

Returns True if command is done, False if still RUNNING

join(*timeout=None*)

Block until command has completed, and return result.

Parameters **timeout** float indicating max seconds to wait for command to complete. Defaults to None.

run()

Run a command.

serialize()

Serializes the AsyncCommandResult into a dict.

Returns dict containing serializable fields in AsyncCommandResult

start()

Begin background execution of command.

class ironic_python_agent.extensions.base.**BaseAgentExtension**(*agent=None*)

Bases: object

check_cmd_presence(*ext_obj*, *ext*, *cmd*)

execute(*command_name*, ***kwargs*)

class ironic_python_agent.extensions.base.**BaseCommandResult**(*command_name*, *command_params*)

Bases: *ironic_python_agent.encoding.SerializableComparable*

Base class for command result.

is_done()

Checks to see if command is still RUNNING.

Returns True if command is done, False if still RUNNING

join()

Returns result of completed command.

serializable_fields = ('id', 'command_name', 'command_params', 'command_stat

wait()

Join the result and extract its value.

Raises if the command failed.

class ironic_python_agent.extensions.base.**ExecuteCommandMixin**
Bases: object

execute_command(*command_name*, ***kwargs*)
Execute an agent command.

get_extension(*extension_name*)

split_command(*command_name*)

class ironic_python_agent.extensions.base.**SyncCommandResult**(*command_name*, *command_params*, *success*, *result_or_error*)
Bases: *ironic_python_agent.extensions.base.BaseCommandResult*

A result from a command that executes synchronously.

ironic_python_agent.extensions.base.**async_command**(*command_name*, *validator=None*)
Will run the command in an AsyncCommandResult in its own thread.

command_name is set based on the func name and command_params will be whatever args/kwargs you pass into the decorated command. Return values of type *str* or *unicode* are prefixed with the *command_name* parameter when returned for consistency.

ironic_python_agent.extensions.base.**get_extension**(*name*)

ironic_python_agent.extensions.base.**init_ext_manager**(*agent*)

ironic_python_agent.extensions.base.**sync_command**(*command_name*, *validator=None*)
Decorate a method to wrap its return value in a SyncCommandResult.

For consistency with @async_command() can also accept a validator which will be used to validate input, although a synchronous command can also choose to implement validation inline.

ironic_python_agent.extensions.clean module

class ironic_python_agent.extensions.clean.**CleanExtension**(*agent=None*)
Bases: *ironic_python_agent.extensions.base.BaseAgentExtension*

execute_clean_step(*step*, *node*, *ports*, *clean_version=None*, ***kwargs*)
Execute a clean step.

Parameters

- **step** A clean step with step, priority and interface keys
- **node** A dict representation of a node
- **ports** A dict representation of ports attached to node
- **clean_version** The clean version as returned by hardware.get_current_versions() at the beginning of cleaning/zapping

Returns a CommandResult object with command_result set to whatever the step returns.

`get_clean_steps`(*node*, *ports*)
Get the list of clean steps supported for the node and ports

Parameters

- **`node`** A dict representation of a node
- **`ports`** A dict representation of ports attached to node

Returns A list of clean steps with keys step, priority, and reboot_requested

ironic_python_agent.extensions.deploy module

class `ironic_python_agent.extensions.deploy.`**`DeployExtension`**(*agent=None*)
Bases: *`ironic_python_agent.extensions.base.BaseAgentExtension`*

`execute_deploy_step`(*step*, *node*, *ports*, *deploy_version=None*, ***kwargs*)
Execute a deploy step.

Parameters

- **`step`** A deploy step with step, priority and interface keys
- **`node`** A dict representation of a node
- **`ports`** A dict representation of ports attached to node
- **`deploy_version`** The deploy version as returned by hardware.get_current_versions() at the beginning of deploying.
- **`kwargs`** The remaining arguments are passed to the step.

Returns a CommandResult object with command_result set to whatever the step returns.

`get_deploy_steps`(*node*, *ports*)
Get the list of deploy steps supported for the node and ports

Parameters

- **`node`** A dict representation of a node
- **`ports`** A dict representation of ports attached to node

Returns A list of deploy steps with keys step, priority, and reboot_requested

ironic_python_agent.extensions.flow module

class `ironic_python_agent.extensions.flow.`**`FlowExtension`**(*agent=None*)
Bases: *`ironic_python_agent.extensions.base.BaseAgentExtension`*, *`ironic_python_agent.extensions.base.ExecuteCommandMixin`*

`start_flow`(*flow=None*)

ironic_python_agent.extensions.image module

class `ironic_python_agent.extensions.image.`**`ImageExtension`**(*agent=None*)

Bases: *`ironic_python_agent.extensions.base.BaseAgentExtension`*

`install_bootloader`(*root_uuid*, *efi_system_part_uuid=None*, *prep_boot_part_uuid=None*, *target_boot_mode='bios'*, *ignore_bootloader_failure=None*)

Install the GRUB2 bootloader on the image.

Parameters

- **`root_uuid`** The UUID of the root partition.
- **`efi_system_part_uuid`** The UUID of the efi system partition. To be used only for uefi boot mode. For uefi boot mode, the boot loader will be installed here.
- **`prep_boot_part_uuid`** The UUID of the PReP Boot partition. Used only for booting ppc64* partition images locally. In this scenario the bootloader will be installed here.
- **`target_boot_mode`** bios, uefi. Only taken into account for softraid, when no efi partition is explicitely provided (happens for whole disk images)

Raises CommandExecutionError if the installation of the bootloader fails.

Raises DeviceNotFound if the root partition is not found.

ironic_python_agent.extensions.iscsi module

class `ironic_python_agent.extensions.iscsi.`**`ISCSIExtension`**(*agent=None*)

Bases: *`ironic_python_agent.extensions.base.BaseAgentExtension`*

`start_iscsi_target`(*iqn=None*, *wipe_disk_metadata=False*, *portal_port=None*)

Expose the disk as an ISCSI target.

Parameters

- **`iqn`** IQN for iSCSI target. If None, a new IQN is generated.
- **`wipe_disk_metadata`** if the disk metadata should be wiped out before the disk is exposed.
- **`portal_port`** customized port for iSCSI port, can be None.

Returns a dict that provides IQN of iSCSI target.

`ironic_python_agent.extensions.iscsi.`**`clean_up`**(*device*)

Clean up iSCSI for a given device.

ironic_python_agent.extensions.log module

class `ironic_python_agent.extensions.log.`**`LogExtension`**(*agent=None*)

Bases: *`ironic_python_agent.extensions.base.BaseAgentExtension`*

`collect_system_logs`()

Collect system logs.

Collect and package diagnostic and support data from the ramdisk.

Raises CommandExecutionError if failed to collect the system logs.

Returns A dictionary with the key *system_logs* and the value of a gzipped and base64 encoded string of the file with the logs.

ironic_python_agent.extensions.poll module

class `ironic_python_agent.extensions.poll.`**`PollExtension`**(*agent=None*)

Bases: *`ironic_python_agent.extensions.base.BaseAgentExtension`*

`get_hardware_info`()

Get the hardware information where IPA is running.

`set_node_info`(*node_info=None*)

Set node lookup data when IPA is running at passive mode.

Parameters **`node_info`** A dictionary contains the information of the node where IPA is running.

ironic_python_agent.extensions.rescue module

class `ironic_python_agent.extensions.rescue.`**`RescueExtension`**(*agent=None*)

Bases: *`ironic_python_agent.extensions.base.BaseAgentExtension`*

`finalize_rescue`(*rescue_password=”*, *hashed=False*)

Sets the rescue password for the rescue user.

`write_rescue_password`(*rescue_password=”*, *hashed=False*)

Write rescue password to a file for use after IPA exits.

Parameters

- **`rescue_password`** Rescue password.
- **`hashed`** Boolean default False indicating if the password being provided is hashed or not. This will be changed in a future version of ironic.

ironic_python_agent.extensions.standby module

class `ironic_python_agent.extensions.standby.`**`ImageDownload`**(*image_info*, *time_obj=None*)

Bases: `object`

Helper class that opens a HTTP connection to download an image.

This class opens a HTTP connection to download an image from a URL and create an iterator so the image can be downloaded in chunks. The MD5 hash of the image being downloaded is calculated on-the-fly.

`verify_image`(*image_location*)

Verifies the checksum of the local images matches expectations.

If this function does not raise ImageChecksumError then it is very likely that the local copy of the image was transmitted and stored correctly.

Parameters **`image_location`** The location of the local image.

Raises ImageChecksumError if the checksum of the local image does not match the checksum as reported by glance in image_info.

class `ironic_python_agent.extensions.standby.`**`StandbyExtension`**(*agent=None*)

Bases: *`ironic_python_agent.extensions.base.BaseAgentExtension`*

Extension which adds stand-by related functionality to agent.

`cache_image`(*image_info=None*, *force=False*)

Asynchronously caches specified image to the local OS device.

Parameters

- **`image_info`** Image information dictionary.
- **`force`** Optional. If True forces cache_image to download and cache image, even if the same image already exists on the local OS install device. Defaults to False.

Raises ImageDownloadError if the image download fails for any reason.

Raises ImageChecksumError if the downloaded images checksum does not match the one reported in image_info.

Raises ImageWriteError if writing the image fails.

`get_partition_uuids`()

Return partition UUIDs.

`power_off`()

Powers off the agents system.

`prepare_image`(*image_info=None*, *configdrive=None*)

Asynchronously prepares specified image on local OS install device.

In this case, prepare means make local machine completely ready to reboot to the image specified by image_info.

Downloads and writes an image to disk if necessary. Also writes a configdrive to disk if the configdrive parameter is specified.

Parameters

- **`image_info`** Image information dictionary.
- **`configdrive`** A string containing the location of the config drive as a URL OR the contents (as gzip/base64) of the configdrive. Optional, defaults to None.

Raises ImageDownloadError if the image download encounters an error.

Raises ImageChecksumError if the checksum of the local image does not match the checksum as reported by glance in image_info.

Raises ImageWriteError if writing the image fails.

Raises InstanceDeployFailure if failed to create config drive. large to store on the given device.

`run_image``()`
Runs image on agents system via reboot.

`sync``()`
Flush file system buffers forcing changed blocks to disk.

Raises CommandExecutionError if flushing file system buffers fails.

Module contents

ironic_python_agent.hardware_managers package

Submodules

ironic_python_agent.hardware_managers.cna module

`class` `ironic_python_agent.hardware_managers.cna.`**`IntelCnaHardwareManager`**
Bases: *`ironic_python_agent.hardware.HardwareManager`*

`HARDWARE_MANAGER_NAME = 'IntelCnaHardwareManager'`

`HARDWARE_MANAGER_VERSION = '1.0'`

`evaluate_hardware_support``()`

ironic_python_agent.hardware_managers.mlnx module

`class` `ironic_python_agent.hardware_managers.mlnx.`**`MellanoxDeviceHardwareManager`**
Bases: *`ironic_python_agent.hardware.HardwareManager`*

Mellanox hardware manager to support a single device

`HARDWARE_MANAGER_NAME = 'MellanoxDeviceHardwareManager'`

`HARDWARE_MANAGER_VERSION = '1'`

`evaluate_hardware_support``()`
Declare level of hardware support provided.

`get_interface_info``(`*`interface_name`*`)`
Return the interface information when its Mellanox and InfiniBand

In case of Mellanox and InfiniBand interface we do the following:

1. Calculate the InfiniBand MAC according to InfiniBand GUID
2. Calculate the client-id according to InfiniBand GUID

Module contents

Submodules

ironic_python_agent.agent module

class `ironic_python_agent.agent.`**`Host`**(*hostname*, *port*)

Bases: `tuple`

`hostname`

Alias for field number 0

`port`

Alias for field number 1

class `ironic_python_agent.agent.`**`IronicPythonAgent`**(*api_url*, *advertise_address*, *listen_address*, *ip_lookup_attempts*, *ip_lookup_sleep*, *network_interface*, *lookup_timeout*, *lookup_interval*, *standalone*, *agent_token*, *hardware_initialization_delay=0*, *advertise_protocol='http'*)

Bases: *`ironic_python_agent.extensions.base.ExecuteCommandMixin`*

Class for base agent functionality.

`force_heartbeat`()

`get_command_result`(*result_id*)

Get a specific command result by ID.

Returns a `ironic_python_agent.extensions.base.BaseCommandResult` object.

Raises RequestedObjectNotFoundError if command with the given ID is not found.

`get_node_uuid`()

Get UUID for Ironic node.

If the agent has not yet heartbeated to Ironic, it will not have the UUID and this will raise an exception.

Returns A string containing the UUID for the Ironic node.

Raises UnknownNodeError if UUID is unknown.

`get_status``()`
Retrieve a serializable status.

Returns a *`ironic_python_agent.agent.IronicPythonAgent`* instance describing the agents status.

`list_command_results``()`
Get a list of command results.

Returns list of `ironic_python_agent.extensions.base.BaseCommandResult` objects.

`process_lookup_data``(`*content*`)`
Update agent configuration from lookup data.

`run``()`
Run the Ironic Python Agent.

`serve_ipa_api``()`
Serve the API until an extension terminates it.

`set_agent_advertise_addr``()`
Set advertised IP address for the agent, if not already set.

If agents advertised IP address is still default (None), try to find a better one. If the agents network interface is None, replace that as well.

Raises LookupAgentIPError if an IP address could not be found

`validate_agent_token``(`*token*`)`

`class` `ironic_python_agent.agent.`**`IronicPythonAgentHeartbeater`**`(`*agent*`)`
Bases: `threading.Thread`

Thread that periodically heartbeats to Ironic.

`do_heartbeat``()`
Send a heartbeat to Ironic.

`force_heartbeat``()`

`max_jitter_multiplier = 0.6`

`min_jitter_multiplier = 0.3`

`run``()`
Start the heartbeat thread.

`stop``()`
Stop the heartbeat thread.

`class` `ironic_python_agent.agent.`**`IronicPythonAgentStatus`**`(`*started_at*, *version*`)`
Bases: *`ironic_python_agent.encoding.Serializable`*

Represents the status of an agent.

`serializable_fields = ('started_at', 'version')`

ironic_python_agent.config module

`ironic_python_agent.config.`**`list_opts`**`()`

`ironic_python_agent.config.`**`override`**`(`*params*`)`

Override configuration with values from a dictionary.

This is used for configuration overrides from mDNS.

Parameters **`params`** new configuration parameters as a dict.

ironic_python_agent.dmi_inspector module

`ironic_python_agent.dmi_inspector.`**`collect_dmidecode_info`**`(`*data*, *failures*`)`

Collect detailed processor, memory and bios info.

The data is gathered using dmidecode utility.

Parameters

- **`data`** mutable dict that well send to inspector
- **`failures`** AccumulatedFailures object

`ironic_python_agent.dmi_inspector.`**`parse_dmi`**`(`*data*`)`

Parse the dmidecode output.

Returns a dict.

ironic_python_agent.encoding module

`class` `ironic_python_agent.encoding.`**`RESTJSONEncoder`**`(`*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*`)`

Bases: `json.encoder.JSONEncoder`

A slightly customized JSON encoder.

`default``(`*o*`)`

Turn an object into a serializable object.

In particular, by calling `Serializable.serialize()` on *o*.

`encode``(`*o*`)`

Turn an object into JSON.

Appends a newline to responses when configured to pretty-print, in order to make use of curl less painful from most shells.

`class` `ironic_python_agent.encoding.`**`Serializable`**

Bases: `object`

Base class for things that can be serialized.

serializable_fields = ()

serialize()

Turn this object into a dict.

class ironic_python_agent.encoding.**SerializableComparable**

Bases: *ironic_python_agent.encoding.Serializable*

A Serializable class which supports some comparison operators

This class supports the __eq__ and __ne__ comparison operators, but intentionally disables the __hash__ operator as some child classes may be mutable. The addition of these comparison operators is mainly used to assist with unit testing.

ironic_python_agent.encoding.**serialize_lib_exc**(*exc*)

Serialize an ironic-lib exception.

ironic_python_agent.errors module

exception ironic_python_agent.errors.**AgentIsBusy**(*command_name*)

Bases: *ironic_python_agent.errors.CommandExecutionError*

message = 'Agent is busy'

status_code = 409

exception ironic_python_agent.errors.**BlockDeviceEraseError**(*details*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when an error occurs erasing a block device.

message = 'Error erasing block device'

exception ironic_python_agent.errors.**BlockDeviceError**(*details*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when a block devices causes an unknown error.

message = 'Block device caused unknown error'

exception ironic_python_agent.errors.**CleaningError**(*details=None*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when a cleaning step fails.

message = 'Clean step failed'

exception ironic_python_agent.errors.**ClockSyncError**(*details=None, *args, **kwargs*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when attempting to sync the system clock.

message = 'Error syncing system clock'

exception ironic_python_agent.errors.**CommandExecutionError**(*details*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when a command fails to execute.

message = 'Command execution failed'

exception ironic_python_agent.errors.**DeploymentError**(*details=None*)
Bases: *ironic_python_agent.errors.RESTError*

Error raised when a deploy step fails.

message = 'Deploy step failed'

exception ironic_python_agent.errors.**DeviceNotFound**(*details*)
Bases: *ironic_python_agent.errors.NotFound*

Error raised when the device to deploy the image onto is not found.

message = 'Error finding the disk or partition device to deploy the image on

exception ironic_python_agent.errors.**ExtensionError**(*details=None*, **args*, ***kwargs*)
Bases: *ironic_python_agent.errors.RESTError*

exception ironic_python_agent.errors.**HardwareManagerMethodNotFound**(*method*)
Bases: *ironic_python_agent.errors.RESTError*

Error raised when all HardwareManagers fail to handle a method.

message = 'No HardwareManager found to handle method'

exception ironic_python_agent.errors.**HardwareManagerNotFound**(*details=None*)
Bases: *ironic_python_agent.errors.RESTError*

Error raised when no valid HardwareManager can be found.

message = 'No valid HardwareManager found'

exception ironic_python_agent.errors.**HeartbeatConflictError**(*details*)
Bases: *ironic_python_agent.errors.IronicAPIError*

ConflictError raised when a heartbeat to the agent API fails.

message = 'ConflictError heartbeating to agent API'

exception ironic_python_agent.errors.**HeartbeatConnectionError**(*details*)
Bases: *ironic_python_agent.errors.IronicAPIError*

Transitory connection failure occured attempting to contact the API.

message = 'Error attempting to heartbeat - Possible transitory network failu

exception ironic_python_agent.errors.**HeartbeatError**(*details*)
Bases: *ironic_python_agent.errors.IronicAPIError*

Error raised when a heartbeat to the agent API fails.

message = 'Error heartbeating to agent API'

exception ironic_python_agent.errors.**ISCSICommandError**(*error_msg*, *exit_code*, *stdout*, *stderr*)
Bases: *ironic_python_agent.errors.ISCSIError*

Error executing TGT command.

exception ironic_python_agent.errors.**ISCSIError**(*error_msg*)
Bases: *ironic_python_agent.errors.RESTError*

Error raised when an image cannot be written to a device.

message = 'Error starting iSCSI target'

exception ironic_python_agent.errors.**ImageChecksumError**(*image_id*, *image_location*, *checksum*, *calculated_checksum*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when an image fails to verify against its checksum.

details_str = 'Image failed to verify against checksum. location: {}; image

message = 'Error verifying image checksum'

exception ironic_python_agent.errors.**ImageDownloadError**(*image_id*, *msg*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when an image cannot be downloaded.

message = 'Error downloading image'

exception ironic_python_agent.errors.**ImageWriteError**(*device*, *exit_code*, *stdout*, *stderr*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when an image cannot be written to a device.

message = 'Error writing image to device'

exception ironic_python_agent.errors.**IncompatibleHardwareMethodError**(*details=None*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when HardwareManager method incompatible with hardware.

message = 'HardwareManager method is not compatible with hardware'

exception ironic_python_agent.errors.**IncompatibleNumaFormatError**(*details=None*, **args*, ***kwargs*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when unexpected format data in NUMA node.

message = 'Error in NUMA node data format'

exception ironic_python_agent.errors.**InspectionError**

Bases: Exception

Failure during inspection.

exception ironic_python_agent.errors.**InvalidCommandError**(*details*)

Bases: *ironic_python_agent.errors.InvalidContentError*

Error which is raised when an unknown command is issued.

message = 'Invalid command'

exception ironic_python_agent.errors.**InvalidCommandParamsError**(*details*)

Bases: *ironic_python_agent.errors.InvalidContentError*

Error which is raised when command parameters are invalid.

message = 'Invalid command parameters'

exception ironic_python_agent.errors.**InvalidContentError**(*details*)
Bases: *ironic_python_agent.errors.RESTError*

Error which occurs when a user supplies invalid content.

Either because that content cannot be parsed according to the advertised *Content-Type*, or due to a content validation error.

message = 'Invalid request body'

status_code = 400

exception ironic_python_agent.errors.**IronicAPIError**(*details*)
Bases: *ironic_python_agent.errors.RESTError*

Error raised when a call to the agent API fails.

message = 'Error in call to ironic-api'

exception ironic_python_agent.errors.**LookupAgentIPError**(*details*)
Bases: *ironic_python_agent.errors.IronicAPIError*

Error raised when automatic IP lookup fails.

message = 'Error finding IP for Ironic Agent'

exception ironic_python_agent.errors.**LookupNodeError**(*details*)
Bases: *ironic_python_agent.errors.IronicAPIError*

Error raised when the node lookup to the Ironic API fails.

message = 'Error getting configuration from Ironic'

exception ironic_python_agent.errors.**NotFound**(*details=None*, **args*, ***kwargs*)
Bases: *ironic_python_agent.errors.RESTError*

Error which occurs if a non-existent API endpoint is called.

details = 'The requested URL was not found.'

message = 'Not found'

status_code = 404

exception ironic_python_agent.errors.**RESTError**(*details=None*, **args*, ***kwargs*)
Bases: Exception, *ironic_python_agent.encoding.Serializable*

Base class for errors generated in ironic-python-client.

details = 'An unexpected error occurred. Please try back later.'

message = 'An error occurred'

serializable_fields = ('type', 'code', 'message', 'details')

status_code = 500

exception ironic_python_agent.errors.**RequestedObjectNotFoundError**(*type_descr*, *obj_id*)

Bases: *ironic_python_agent.errors.NotFound*

exception ironic_python_agent.errors.**SoftwareRAIDError**(*details*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when a Software RAID causes an error.

message = 'Software RAID caused unknown error'

exception ironic_python_agent.errors.**SystemRebootError**(*exit_code*, *stdout*, *stderr*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when a system cannot reboot.

message = 'Error rebooting system'

exception ironic_python_agent.errors.**UnknownNodeError**(*details=None*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when the agent is not associated with an Ironic node.

message = 'Agent is not associated with an Ironic node'

exception ironic_python_agent.errors.**VersionMismatch**(*agent_version*, *node_version*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when Ironic and the Agent have different versions.

If the agent version has changed since get_clean_steps or get_deploy_steps was called by the Ironic conductor, it indicates the agent has been updated (either on purpose, or a new agent was deployed and the node was rebooted). Since we cannot know if the upgraded IPA will work with cleaning/deploy as it stands (steps could have different priorities, either in IPA or in other Ironic interfaces), we should restart the process from the start.

message = 'Hardware managers version mismatch, reload agent with correct ver

exception ironic_python_agent.errors.**VirtualMediaBootError**(*details*)

Bases: *ironic_python_agent.errors.RESTError*

Error raised when virtual media device cannot be found for config.

message = 'Configuring agent from virtual media failed'

ironic_python_agent.hardware module

class ironic_python_agent.hardware.**BlockDevice**(*name*, *model*, *size*, *rotational*, *wwn=None*, *serial=None*, *vendor=None*, *wwn_with_extension=None*, *wwn_vendor_extension=None*, *hctl=None*, *by_path=None*)

Bases: *ironic_python_agent.encoding.SerializableComparable*

serializable_fields = ('name', 'model', 'size', 'rotational', 'wwn', 'serial

class ironic_python_agent.hardware.**BootInfo**(*current_boot_mode*, *pxe_interface=None*)
Bases: *ironic_python_agent.encoding.SerializableComparable*

serializable_fields = ('current_boot_mode', 'pxe_interface')

class ironic_python_agent.hardware.**CPU**(*model_name*, *frequency*, *count*, *architecture*, *flags=None*)
Bases: *ironic_python_agent.encoding.SerializableComparable*

serializable_fields = ('model_name', 'frequency', 'count', 'architecture', '

class ironic_python_agent.hardware.**GenericHardwareManager**
Bases: *ironic_python_agent.hardware.HardwareManager*

HARDWARE_MANAGER_NAME = 'generic_hardware_manager'

HARDWARE_MANAGER_VERSION = '1.1'

apply_configuration(*node*, *ports*, *raid_config*, *delete_existing=True*)
Apply RAID configuration.

Parameters

- **node** A dictionary of the node object.
- **ports** A list of dictionaries containing information of ports for the node.
- **raid_config** The configuration to apply.
- **delete_existing** Whether to delete the existing configuration.

collect_lldp_data(*interface_names*)
Collect and convert LLDP info from the node.

In order to process the LLDP information later, the raw data needs to be converted for serialization purposes.

Parameters **interface_names** list of names of nodes interfaces.

Returns a dict, containing the lldp data from every interface.

create_configuration(*node*, *ports*)
Create a RAID configuration.

Unless overwritten by a local hardware manager, this method will create a software RAID configuration as read from the nodes target_raid_config.

Parameters

- **node** A dictionary of the node object.
- **ports** A list of dictionaries containing information of ports for the node.

Returns The current RAID configuration in the usual format.

Raises SoftwareRAIDError if the desired configuration is not valid or if there was an error when creating the RAID devices.

delete_configuration(*node*, *ports*)
Delete a RAID configuration.

Unless overwritten by a local hardware manager, this method will delete all software RAID devices on the node. NOTE(arne_wiebalck): It may be worth considering to only delete

RAID devices in the nodes target_raid_config. If that config has been lost, though, the cleanup may become difficult. So, for now, we delete everything we detect.

Parameters

- **`node`** A dictionary of the node object
- **`ports`** A list of dictionaries containing information of ports for the node

`erase_block_device`(*node*, *block_device*)

Attempt to erase a block device.

Implementations should detect the type of device and erase it in the most appropriate way possible. Generic implementations should support common erase mechanisms such as ATA secure erase, or multi-pass random writes. Operators with more specific needs should override this method in order to detect and handle interesting cases, or delegate to the parent class to handle generic cases.

For example: operators running ACME MagicStore (TM) cards alongside standard SSDs might check whether the device is a MagicStore and use a proprietary tool to erase that, otherwise call this method on their parent class. Upstream submissions of common functionality are encouraged.

This interface could be called concurrently to speed up erasure, as such, it should be implemented in a thread-safe way.

Parameters

- **`node`** Ironic node object
- **`block_device`** a BlockDevice indicating a device to be erased.

Raises

- ***`IncompatibleHardwareMethodError`*** when there is no known way to erase the block device
- ***`BlockDeviceEraseError`*** when there is an error erasing the block device

`erase_devices_metadata`(*node*, *ports*)

Attempt to erase the disk devices metadata.

Parameters

- **`node`** Ironic node object
- **`ports`** list of Ironic port objects

Raises ***`BlockDeviceEraseError`*** when theres an error erasing the block device

`evaluate_hardware_support`()

`generate_tls_certificate`(*ip_address*)

Generate a TLS certificate for the IP address.

`get_bios_given_nic_name`(*interface_name*)

Collect the BIOS given NICs name.

This function uses the biosdevname utility to collect the BIOS given name of network interfaces.

The collected data is added to the network interface inventory with an extra field named `biosdevname`.

Parameters **`interface_name`** list of names of nodes interfaces.

Returns the BIOS given NIC name of nodes interfaces or default as None.

`get_bmc_address`()

Attempt to detect BMC IP address

Returns IP address of lan channel or 0.0.0.0 in case none of them is configured properly

`get_bmc_v6address`()

Attempt to detect BMC v6 address

Returns IPv6 address of lan channel or ::/0 in case none of them is configured properly. May return None value if it cannot interract with system tools or critical error occurs.

`get_boot_info`()

`get_clean_steps`(*node*, *ports*)

Get a list of clean steps with priority.

Returns a list of steps. Each step is represented by a dict:

```
{
 'interface': the name of the driver interface that should execute
              the step.
 'step': the HardwareManager function to call.
 'priority': the order steps will be run in. Ironic will sort all
             the clean steps from all the drivers, with the␣
↪largest
             priority step being run first. If priority is set to␣
↪0,
             the step will not be run during cleaning, but may be
             run during zapping.
 'reboot_requested': Whether the agent should request Ironic␣
↪reboots
                     the node via the power driver after the
                     operation completes.
 'abortable': Boolean value. Whether the clean step can be
              stopped by the operator or not. Some clean step may
              cause non-reversible damage to a machine if␣
↪interrupted
              (i.e firmware update), for such steps this parameter
              should be set to False. If no value is set for this
              parameter, Ironic will consider False (non-
↪abortable).
}
```

If multiple hardware managers return the same step name, the following logic will be used to determine which managers step wins:

- Keep the step that belongs to HardwareManager with highest HardwareSupport (larger int) value.
- If equal support level, keep the step with the higher defined priority (larger int).

- If equal support level and priority, keep the step associated with the HardwareManager whose name comes earlier in the alphabet.

The steps will be called using *hardware.dispatch_to_managers* and handled by the best suited hardware manager. If you need a step to be executed by only your hardware manager, ensure it has a unique step name.

node and *ports* can be used by other hardware managers to further determine if a clean step is supported for the node.

Parameters

- **node** Ironic node object
- **ports** list of Ironic port objects

Returns a list of cleaning steps, where each step is described as a dict as defined above

get_cpus()

get_deploy_steps(*node*, *ports*)

Get a list of deploy steps with priority.

Returns a list of steps. Each step is represented by a dict:

```
{
 'interface': the name of the driver interface that should execute
              the step.
 'step': the HardwareManager function to call.
 'priority': the order steps will be run in. Ironic will sort all
             the deploy steps from all the drivers, with the␣
↪largest
             priority step being run first. If priority is set to␣
↪0,
             the step will not be run during deployment
             automatically, but may be requested via deploy
             templates.
 'reboot_requested': Whether the agent should request Ironic␣
↪reboots
                     the node via the power driver after the
                     operation completes.
 'argsinfo': arguments specification.
}
```

If multiple hardware managers return the same step name, the following logic will be used to determine which managers step wins:

- Keep the step that belongs to HardwareManager with highest HardwareSupport (larger int) value.
- If equal support level, keep the step with the higher defined priority (larger int).
- If equal support level and priority, keep the step associated with the HardwareManager whose name comes earlier in the alphabet.

The steps will be called using *hardware.dispatch_to_managers* and handled by the best suited hardware manager. If you need a step to be executed by only your hardware manager, ensure it has a unique step name.

node and *ports* can be used by other hardware managers to further determine if a deploy step is supported for the node.

Parameters

- **node** Ironic node object
- **ports** list of Ironic port objects

Returns a list of deploying steps, where each step is described as a dict as defined above

get_interface_info(*interface_name*)

get_ipv4_addr(*interface_id*)

get_ipv6_addr(*interface_id*)
Get the default IPv6 address assigned to the interface.

With different networking environment, the address could be a link-local address, ULA or something else.

get_memory()

get_os_install_device(*permit_refresh=False*)

get_system_vendor_info()

list_block_devices(*include_partitions=False*)
List physical block devices

Parameters **include_partitions** If to include partitions

Returns A list of BlockDevices

list_network_interfaces()

validate_configuration(*raid_config*, *node*)
Validate a (software) RAID configuration

Validate a given raid_config, in particular with respect to the limitations of the current implementation of software RAID support.

Parameters **raid_config** The current RAID configuration in the usual format.

write_image(*node*, *ports*, *image_info*, *configdrive=None*)
A deploy step to write an image.

Downloads and writes an image to disk if necessary. Also writes a configdrive to disk if the configdrive parameter is specified.

Parameters

- **node** A dictionary of the node object
- **ports** A list of dictionaries containing information of ports for the node
- **image_info** Image information dictionary.
- **configdrive** A string containing the location of the config drive as a URL OR the contents (as gzip/base64) of the configdrive. Optional, defaults to None.

class `ironic_python_agent.hardware.`**`HardwareManager`**

Bases: `object`

`erase_block_device`(*node*, *block_device*)

Attempt to erase a block device.

Implementations should detect the type of device and erase it in the most appropriate way possible. Generic implementations should support common erase mechanisms such as ATA secure erase, or multi-pass random writes. Operators with more specific needs should override this method in order to detect and handle interesting cases, or delegate to the parent class to handle generic cases.

For example: operators running ACME MagicStore (TM) cards alongside standard SSDs might check whether the device is a MagicStore and use a proprietary tool to erase that, otherwise call this method on their parent class. Upstream submissions of common functionality are encouraged.

This interface could be called concurrently to speed up erasure, as such, it should be implemented in a thread-safe way.

Parameters

- **`node`** Ironic node object
- **`block_device`** a BlockDevice indicating a device to be erased.

Raises

- *`IncompatibleHardwareMethodError`* when there is no known way to erase the block device
- *`BlockDeviceEraseError`* when there is an error erasing the block device

`erase_devices`(*node*, *ports*)

Erase any device that holds user data.

By default this will attempt to erase block devices. This method can be overridden in an implementation-specific hardware manager in order to erase additional hardware, although backwards-compatible upstream submissions are encouraged.

Parameters

- **`node`** Ironic node object
- **`ports`** list of Ironic port objects

Returns a dictionary in the form {device.name: erasure output}

abstract **`evaluate_hardware_support`**()

`generate_tls_certificate`(*ip_address*)

`get_bmc_address`()

`get_bmc_v6address`()

`get_boot_info`()

`get_clean_steps`(*node*, *ports*)

Get a list of clean steps with priority.

Returns a list of steps. Each step is represented by a dict:

```
{
 'interface': the name of the driver interface that should execute
              the step.
 'step': the HardwareManager function to call.
 'priority': the order steps will be run in. Ironic will sort all
             the clean steps from all the drivers, with the␣
↪largest
             priority step being run first. If priority is set to␣
↪0,
             the step will not be run during cleaning, but may be
             run during zapping.
 'reboot_requested': Whether the agent should request Ironic␣
↪reboots
                     the node via the power driver after the
                     operation completes.
 'abortable': Boolean value. Whether the clean step can be
              stopped by the operator or not. Some clean step may
              cause non-reversible damage to a machine if␣
↪interrupted
              (i.e firmware update), for such steps this parameter
              should be set to False. If no value is set for this
              parameter, Ironic will consider False (non-
↪abortable).
}
```

If multiple hardware managers return the same step name, the following logic will be used to determine which managers step wins:

- Keep the step that belongs to HardwareManager with highest HardwareSupport (larger int) value.
- If equal support level, keep the step with the higher defined priority (larger int).
- If equal support level and priority, keep the step associated with the HardwareManager whose name comes earlier in the alphabet.

The steps will be called using *hardware.dispatch_to_managers* and handled by the best suited hardware manager. If you need a step to be executed by only your hardware manager, ensure it has a unique step name.

node and *ports* can be used by other hardware managers to further determine if a clean step is supported for the node.

Parameters

- **`node`** Ironic node object
- **`ports`** list of Ironic port objects

Returns a list of cleaning steps, where each step is described as a dict as defined above

`get_cpus`()

`get_deploy_steps`(*node*, *ports*)

Get a list of deploy steps with priority.

Returns a list of steps. Each step is represented by a dict:

```
{
 'interface': the name of the driver interface that should execute
              the step.
 'step': the HardwareManager function to call.
 'priority': the order steps will be run in. Ironic will sort all
             the deploy steps from all the drivers, with the␣
↪largest
             priority step being run first. If priority is set to␣
↪0,
             the step will not be run during deployment
             automatically, but may be requested via deploy
             templates.
 'reboot_requested': Whether the agent should request Ironic␣
↪reboots
                     the node via the power driver after the
                     operation completes.
 'argsinfo': arguments specification.
}
```

If multiple hardware managers return the same step name, the following logic will be used to determine which managers step wins:

- Keep the step that belongs to HardwareManager with highest HardwareSupport (larger int) value.
- If equal support level, keep the step with the higher defined priority (larger int).
- If equal support level and priority, keep the step associated with the HardwareManager whose name comes earlier in the alphabet.

The steps will be called using *hardware.dispatch_to_managers* and handled by the best suited hardware manager. If you need a step to be executed by only your hardware manager, ensure it has a unique step name.

node and *ports* can be used by other hardware managers to further determine if a deploy step is supported for the node.

Parameters

- **`node`** Ironic node object
- **`ports`** list of Ironic port objects

Returns a list of deploying steps, where each step is described as a dict as defined above

`get_interface_info`(*interface_name*)

`get_memory`()

`get_os_install_device`(*permit_refresh=False*)

`get_version`()

Get a name and version for this hardware manager.

In order to avoid errors and make agent upgrades painless, cleaning will check the version of all hardware managers during get_clean_steps at the beginning of cleaning and before executing each step in the agent.

The agent isnt aware of the steps being taken before or after via out of band steps, so it can never know if a new step is safe to run. Therefore, we default to restarting the whole process.

Returns a dictionary with two keys: *name* and *version*, where *name* is a string identifying the hardware manager and *version* is an arbitrary version string. *name* will be a class variable called HARDWARE_MANAGER_NAME, or default to the class name and *version* will be a class variable called HARDWARE_MANAGER_VERSION or default to 1.0.

`list_block_devices`(*include_partitions=False*)
List physical block devices

Parameters **`include_partitions`** If to include partitions

Returns A list of BlockDevices

`list_hardware_info`()
Return full hardware inventory as a serializable dict.

This inventory is sent to Ironic on lookup and to Inspector on inspection.

Returns a dictionary representing inventory

`list_network_interfaces`()

`wait_for_disks`()
Wait for the root disk to appear.

Wait for at least one suitable disk to show up or a specific disk if any device hint is specified. Otherwise neither inspection not deployment have any chances to succeed.

class `ironic_python_agent.hardware.`**`HardwareSupport`**
Bases: `object`

Example priorities for hardware managers.

Priorities for HardwareManagers are integers, where largest means most specific and smallest means most generic. These values are guidelines that suggest values that might be returned by calls to *evaluate_hardware_support()*. No HardwareManager in mainline IPA will ever return a value greater than MAINLINE. Third party hardware managers should feel free to return values of SERVICE_PROVIDER or greater to distinguish between additional levels of hardware support.

`GENERIC = 1`

`MAINLINE = 2`

`NONE = 0`

`SERVICE_PROVIDER = 3`

class `ironic_python_agent.hardware.`**`HardwareType`**
Bases: `object`

`MAC_ADDRESS = 'mac_address'`

class `ironic_python_agent.hardware.`**`Memory`**(*total, physical_mb=None*)
Bases: *`ironic_python_agent.encoding.SerializableComparable`*

`serializable_fields = ('total', 'physical_mb')`

class ironic_python_agent.hardware.**NetworkInterface**(*name*, *mac_addr*, *ipv4_address=None*, *ipv6_address=None*, *has_carrier=True*, *lldp=None*, *vendor=None*, *product=None*, *client_id=None*, *biosdevname=None*)

Bases: *ironic_python_agent.encoding.SerializableComparable*

serializable_fields = ('name', 'mac_address', 'ipv4_address', 'ipv6_address'

class ironic_python_agent.hardware.**SystemVendorInfo**(*product_name*, *serial_number*, *manufacturer*)

Bases: *ironic_python_agent.encoding.SerializableComparable*

serializable_fields = ('product_name', 'serial_number', 'manufacturer')

ironic_python_agent.hardware.**cache_node**(*node*)

Store the node object in the hardware module.

Stores the node object in the hardware module to facilitate the access of a node information in the hardware extensions.

If the new node does not match the previously cached one, wait for the expected root device to appear.

Parameters

- **node** Ironic node object
- **wait_for_disks** Default True switch to wait for disk setup to be completed so the node information can be aligned with the physical storage devices of the host. This is likely to be used in unit testing.

ironic_python_agent.hardware.**check_versions**(*provided_version=None*)

Ensure the version of hardware managers hasnt changed.

Parameters **provided_version** Hardware manager versions used by ironic.

Raises errors.VersionMismatch if any hardware manager version on the currently running agent doesnt match the one stored in provided_version.

Returns None

ironic_python_agent.hardware.**deduplicate_steps**(*candidate_steps*)

Remove duplicated clean or deploy steps

Deduplicates steps returned from HardwareManagers to prevent running a given step more than once. Other than individual step priority, it doesnt actually impact the deployment which specific steps are kept and what HardwareManager they are associated with. However, in order to make testing easier, this method returns deterministic results.

Uses the following filtering logic to decide which step wins:

- Keep the step that belongs to HardwareManager with highest HardwareSupport (larger int) value.

- If equal support level, keep the step with the higher defined priority (larger int).
- If equal support level and priority, keep the step associated with the HardwareManager whose name comes earlier in the alphabet.

Parameters **`candidate_steps`** A dict containing all possible steps from all managers, key=manager, value=list of steps

Returns A deduplicated dictionary of {hardware_manager: [steps]}

`ironic_python_agent.hardware.`**`dispatch_to_all_managers`**`(`*method*, **args*, ***kwargs*`)`

Dispatch a method to all hardware managers.

Dispatches the given method in priority order as sorted by *get_managers*. If the method doesnt exist or raises IncompatibleHardwareMethodError, it continues to the next hardware manager. All managers that have hardware support for this node will be called, and their responses will be added to a dictionary of the form {HardwareManagerClassName: response}.

Parameters

- **`method`** hardware manager method to dispatch
- **`args`** arguments to dispatched method
- **`kwargs`** keyword arguments to dispatched method

Raises *`errors.HardwareManagerMethodNotFound`* if all managers raise IncompatibleHardwareMethodError.

Returns a dictionary with keys for each hardware manager that returns a response and the value as a list of results from that hardware manager.

`ironic_python_agent.hardware.`**`dispatch_to_managers`**`(`*method*, **args*, ***kwargs*`)`

Dispatch a method to best suited hardware manager.

Dispatches the given method in priority order as sorted by *get_managers*. If the method doesnt exist or raises IncompatibleHardwareMethodError, it is attempted again with a more generic hardware manager. This continues until a method executes that returns any result without raising an IncompatibleHardwareMethodError.

Parameters

- **`method`** hardware manager method to dispatch
- **`args`** arguments to dispatched method
- **`kwargs`** keyword arguments to dispatched method

Returns result of successful dispatch of method

Raises

- *`HardwareManagerMethodNotFound`* if all managers failed the method
- *`HardwareManagerNotFound`* if no valid hardware managers found

`ironic_python_agent.hardware.`**`get_cached_node`**`()`

Guard function around the module variable NODE.

`ironic_python_agent.hardware.`**`get_current_versions`**`()`

Fetches versions from all hardware managers.

Returns Dict in the format {name: version} containing one entry for every hardware manager.

`ironic_python_agent.hardware.`**`get_holder_disks`**`(`*raid_device*`)`

Get the holder disks of a Software RAID device.

Examine an md device and return its underlying disks.

Parameters **`raid_device`** A Software RAID block device name.

Returns A list of the holder disks.

`ironic_python_agent.hardware.`**`get_managers`**`()`

Get a list of hardware managers in priority order.

Use stevedore to find all eligible hardware managers, sort them based on self-reported (via evaluate_hardware_support()) priorities, and return them in a list. The resulting list is cached in _global_managers.

Returns Priority-sorted list of hardware managers

Raises *`HardwareManagerNotFound`* if no valid hardware managers found

`ironic_python_agent.hardware.`**`is_md_device`**`(`*raid_device*`)`

Check if a device is an md device

Check if a device is a Software RAID (md) device.

Parameters **`raid_device`** A Software RAID block device name.

Returns True if the device is an md device, False otherwise.

`ironic_python_agent.hardware.`**`list_all_block_devices`**`(`*block_type='disk', ignore_raid=False, ignore_floppy=True, ignore_empty=True*`)`

List all physical block devices

The switches we use for lsblk: P for KEY=value output, b for size output in bytes, i to ensure ascii characters only, and o to specify the fields/columns we need.

Broken out as its own function to facilitate custom hardware managers that dont need to subclass GenericHardwareManager.

Parameters

- **`block_type`** Type of block device to find
- **`ignore_raid`** Ignore auto-identified raid devices, example: md0 Defaults to false as these are generally disk devices and should be treated as such if encountered.
- **`ignore_floppy`** Ignore floppy disk devices in the block device list. By default, these devices are filtered out.
- **`ignore_empty`** Whether to ignore disks with size equal 0.

Returns A list of BlockDevices

`ironic_python_agent.hardware.`**`list_hardware_info`**`(`*use_cache=True*`)`
List hardware information with caching.

`ironic_python_agent.hardware.`**`md_get_raid_devices`**`()`
Get all discovered Software RAID (md) devices

Returns A python dict containing details about the discovered RAID devices

`ironic_python_agent.hardware.`**`md_restart`**`(`*raid_device*`)`
Restart an md device

Stop and re-assemble a Software RAID (md) device.

Parameters **`raid_device`** A Software RAID block device name.

Raises CommandExecutionError in case the restart fails.

`ironic_python_agent.hardware.`**`save_api_client`**`(`*client=None*, *timeout=None*, *interval=None*`)`
Preserves access to the API client for potential later re-use.

`ironic_python_agent.hardware.`**`update_cached_node`**`()`
Attmepts to update the node cache via the API

ironic_python_agent.inspect module

`class` `ironic_python_agent.inspect.`**`IronicInspection`**
Bases: `threading.Thread`

Class for manual inspection functionality.

`backoff_factor = 2.7`

`max_delay = 1200`

`max_jitter_multiplier = 1.2`

`min_jitter_multiplier = 0.7`

`run``()`
Run Inspection.

ironic_python_agent.inspector module

`ironic_python_agent.inspector.`**`call_inspector`**`(`*data*, *failures*`)`
Post data to inspector.

`ironic_python_agent.inspector.`**`collect_default`**`(`*data*, *failures*`)`
The default inspection collector.

This is the only collector that is called by default. It collects the whole inventory as returned by the hardware manager(s).

It also tries to get BMC address, PXE boot device and the expected root device.

Parameters

- **`data`** mutable data that well send to inspector

- **failures** AccumulatedFailures object

`ironic_python_agent.inspector.`**`collect_extra_hardware`**`(`*data*, *failures*`)`

Collect detailed inventory using hardware-detect utility.

Recognizes ipa-inspection-benchmarks with list of benchmarks (possible values are cpu, disk, mem) to run. No benchmarks are run by default, as theyre pretty time-consuming.

Puts collected data as JSON under data key. Requires hardware python package to be installed on the ramdisk in addition to the packages in requirements.txt.

Parameters

- **data** mutable data that well send to inspector
- **failures** AccumulatedFailures object

`ironic_python_agent.inspector.`**`collect_logs`**`(`*data*, *failures*`)`

Collect system logs from the ramdisk.

As inspection runs before any nodes details are known, its handy to have logs returned with data. This collector sends logs to inspector in format expected by the ramdisk_error plugin: base64 encoded tar.gz.

This collector should be installed last in the collector chain, otherwise it wont collect enough logs.

This collector does not report failures.

Parameters

- **data** mutable data that well send to inspector
- **failures** AccumulatedFailures object

`ironic_python_agent.inspector.`**`collect_pci_devices_info`**`(`*data*, *failures*`)`

Collect a list of PCI devices.

Each PCI device entry in list is a dictionary containing vendor_id and product_id keys, which will be then used by the ironic inspector to distinguish various PCI devices.

The data is gathered from /sys/bus/pci/devices directory.

Parameters

- **data** mutable data that well send to inspector
- **failures** AccumulatedFailures object

`ironic_python_agent.inspector.`**`extension_manager`**`(`*names*`)`

`ironic_python_agent.inspector.`**`inspect`**`()`

Optionally run inspection on the current node.

If `inspection_callback_url` is set in the configuration, get the hardware inventory from the node and post it back to the inspector.

Returns node UUID if inspection was successful, None if associated node was not found in inspector cache. None is also returned if inspector support is not enabled.

`ironic_python_agent.inspector.`**`wait_for_dhcp`**`()`

Wait until NICs get their IP addresses via DHCP or timeout happens.

Depending on the value of inspection_dhcp_all_interfaces configuration option will wait for either all or only PXE booting NIC.

Note: only supports IPv4 addresses for now.

Returns True if all NICs got IP addresses, False if timeout happened. Also returns True if waiting is disabled via configuration.

ironic_python_agent.ironic_api_client module

class `ironic_python_agent.ironic_api_client.`**`APIClient`**(*api_url*)

Bases: `object`

`agent_token = None`

`api_version = 'v1'`

`heartbeat`(*uuid*, *advertise_address*, *advertise_protocol='http'*, *generated_cert=None*)

`heartbeat_api = '/v1/heartbeat/{uuid}'`

`lookup_api = '/v1/lookup'`

`lookup_node`(*hardware_info*, *timeout*, *starting_interval*, *node_uuid=None*, *max_interval=30*)

`supports_auto_tls`()

ironic_python_agent.netutils module

class `ironic_python_agent.netutils.`**`RawPromiscuousSockets`**(*interface_names*, *protocol*)

Bases: `object`

`ironic_python_agent.netutils.`**`bring_up_vlan_interfaces`**(*interfaces_list*)

Bring up vlan interfaces based on kernel params

Use the configured value of `enable_vlan_interfaces` to determine if VLAN interfaces should be brought up using `ip` commands. If `enable_vlan_interfaces` defines a particular vlan then bring up that vlan. If it defines an interface or `all` then use LLDP info to figure out which VLANs should be brought up.

Parameters **`interfaces_list`** List of current interfaces

Returns List of vlan interface names that have been added

`ironic_python_agent.netutils.`**`get_default_ip_addr`**(*type*, *interface_id*)

Retrieve default IPv4 or IPv6 address.

`ironic_python_agent.netutils.`**`get_hostname`**()

`ironic_python_agent.netutils.`**`get_ipv4_addr`**(*interface_id*)

`ironic_python_agent.netutils.`**`get_ipv6_addr`**(*interface_id*)

`ironic_python_agent.netutils.`**`get_lldp_info`**(*interface_names*)

Get LLDP info from the switch(es) the agent is connected to.

Listens on either a single or all interfaces for LLDP packets, then parses them. If no LLDP packets are received before lldp_timeout, returns a dictionary in the form {interface: [],}.

Parameters **`interface_names`** The interface to listen for packets on. If None, will listen on each interface.

Returns A dictionary in the form {interface: [(lldp_type, lldp_data)],}

`ironic_python_agent.netutils.`**`get_mac_addr`**(*interface_id*)

`ironic_python_agent.netutils.`**`get_wildcard_address`**()

`class` `ironic_python_agent.netutils.`**`ifreq`**

Bases: `_ctypes.Structure`

Class for setting flags on a socket.

`ifr_flags`

Structure/Union member

`ifr_ifrn`

Structure/Union member

`ironic_python_agent.netutils.`**`interface_has_carrier`**(*interface_name*)

`ironic_python_agent.netutils.`**`wrap_ipv6`**(*ip*)

ironic_python_agent.numa_inspector module

`ironic_python_agent.numa_inspector.`**`collect_numa_topology_info`**(*data, failures*)

Collect the NUMA topology information.

The data is gathered from /sys/devices/system/node/node<X> and /sys/class/net/ directories. The information is collected in the form of:

```
{
  "numa_topology": {
    "ram": [{"numa_node": <numa_node_id>, "size_kb": <memory_in_kb>},
            ...],
    "cpus": [
      {
        "cpu": <cpu_id>, "numa_node": <numa_node_id>,
        "thread_siblings": [<list of sibling threads>]
      },
      ...,
    ],
    "nics": [
      {"name": "<network interface name>", "numa_node": <numa_node_id>
↪},
      ...,
    ]
  }
}
```

Parameters

- **`data`** mutable data that well send to inspector
- **`failures`** AccumulatedFailures object

Returns None

`ironic_python_agent.numa_inspector.`**`get_nodes_cores_info`**`(`*numa_node_dirs*`)`

Collect the NUMA nodes cpus and threads information.

NUMA nodes path: /sys/devices/system/node/node<node_id>

Thread dirs path: /sys/devices/system/node/node<node_id>/cpu<thread_id>

CPU id file path: /sys/devices/system/node/node<node_id>/cpu<thread_id>/
topology/core_id

The information is returned in the form of:

```
"cpus": [
      {
        "cpu": <cpu_id>, "numa_node": <numa_node_id>,
        "thread_siblings": [<list of sibling threads>]
      },
      ...,
    ]
```

Parameters **`numa_node_dirs`** A list of NUMA node directories

Raises IncompatibleNumaFormatError: when unexpected format data in NUMA node

Returns A list of cpu information with NUMA node id and thread siblings

`ironic_python_agent.numa_inspector.`**`get_nodes_memory_info`**`(`*numa_node_dirs*`)`

Collect the NUMA nodes memory information.

The information is returned in the form of:

```
"ram": [{"numa_node": <numa_node_id>, "size_kb": <memory_in_kb>}, ...]
```

Parameters **`numa_node_dirs`** A list of NUMA node directories

Raises IncompatibleNumaFormatError: when unexpected format data in NUMA node

Returns A list of memory information with NUMA node id

`ironic_python_agent.numa_inspector.`**`get_nodes_nics_info`**`(`*nic_device_path*`)`

Collect the NUMA nodes nics information.

The information is returned in the form of:

```
"nics": [
      {"name": "<network interface name>",
       "numa_node": <numa_node_id>},
      ...,
    ]
```

Parameters **`nic_device_path`** nic device directory path

Raises IncompatibleNumaFormatError: when unexpected format data in NUMA node

Returns A list of nics information with NUMA node id

`ironic_python_agent.numa_inspector.`**`get_numa_node_id`**`(`*numa_node_dir*`)`
Provides the NUMA node id from NUMA node directory

Parameters **`numa_node_dir`** NUMA node directory

Raises IncompatibleNumaFormatError: when unexpected format data in NUMA node dir

Returns NUMA node id

ironic_python_agent.raid_utils module

`ironic_python_agent.raid_utils.`**`calc_raid_partition_sectors`**`(`*psize*, *start*`)`
Calculates end sector and converts start and end sectors including

the unit of measure, compatible with parted. :param psize: size of the raid partition :param start: start sector of the raid partion in integer format :return: start and end sector in parted compatible format, end sector

as integer

`ironic_python_agent.raid_utils.`**`calculate_raid_start`**`(`*target_boot_mode*, *partition_table_type*, *dev_name*`)`
Define the start sector for the raid partition.

Parameters

- **`target_boot_mode`** the node boot mode.
- **`partition_table_type`** the node partition label, gpt or msdos.
- **`dev_name`** block device in the raid configuration.

Returns The start sector for the raid partition.

`ironic_python_agent.raid_utils.`**`create_raid_partition_tables`**`(`*block_devices*, *partition_table_type*, *target_boot_mode*`)`
Creates partition tables in all disks in a RAID configuration and

reports the starting sector for each partition on each disk. :param block_devices: disks where we want to create the partition tables. :param partition_table_type: type of partition table to create, for example

gpt or msdos.

Parameters **`target_boot_mode`** the node selected boot mode, for example uefi or bios.

Returns a dictionary of devices and the start of the corresponding partition.

`ironic_python_agent.raid_utils.`**`get_block_devices_for_raid`**(*block_devices*, *logical_disks*)

Get block devices that are involved in the RAID configuration.

This call does two things: * Collect all block devices that are involved in RAID. * Update each logical disks with suitable block devices.

ironic_python_agent.tls_utils module

`class` `ironic_python_agent.tls_utils.`**`TlsCertificate`**(*text*, *path*, *private_key_path*)

Bases: `tuple`

`path`
Alias for field number 1

`private_key_path`
Alias for field number 2

`text`
Alias for field number 0

`ironic_python_agent.tls_utils.`**`generate_tls_certificate`**(*ip_address*, *common_name=None*, *valid_for_days=90*)

Generate a self-signed TLS certificate.

Parameters

- **`ip_address`** IP address the certificate will be valid for.
- **`common_name`** Content for the common name field (e.g. host name). Defaults to the current host name.
- **`valid_for_days`** Number of days the certificate will be valid for.

Returns a TlsCertificate object.

ironic_python_agent.utils module

`class` `ironic_python_agent.utils.`**`AccumulatedFailures`**(*exc_class=<class 'RuntimeError'>*)

Bases: `object`

Object to accumulate failures without raising exception.

`add`(*fail*, **fmt*)
Add failure with optional formatting.

Parameters

- **`fail`** exception or error string
- **`fmt`** formatting arguments (only if fail is a string)

get_error()
Get error string or None.

raise_if_needed()
Raise exception if error list is not empty.

Raises RuntimeError

ironic_python_agent.utils.**collect_system_logs**(*journald_max_lines=None*)
Collect system logs.

Collect system logs, for distributions using systemd the logs will come from journald. On other distributions the logs will come from the /var/log directory and dmesg output.

Parameters **journald_max_lines** Maximum number of lines to retrieve from the journald. if None, return everything.

Returns A tar, gzip base64 encoded string with the logs.

ironic_python_agent.utils.**create_partition_table**(*dev_name*, *partition_table_type*)
Create a partition table on a disk using parted.

Parameters

- **dev_name** the disk where we want to create the partition table.
- **partition_table_type** the type of partition table we want to create, for example gpt or msdos.

Raises CommandExecutionError if an error is encountered while attempting to create the partition table.

ironic_python_agent.utils.**determine_time_method**()
Helper method to determine what time utility is present.

Returns ntpdate if ntpdate has been found, chrony if chrony was located, and None if neither are located. If both tools are present, chrony will supercede ntpdate.

ironic_python_agent.utils.**execute**(**cmd*, ***kwargs*)
Convenience wrapper around ironic_libs execute() method.

Executes and logs results from a system command.

ironic_python_agent.utils.**extract_device**(*part*)
Extract the device from a partition name or path.

Parameters **part** the partition

Returns a device if success, None otherwise

ironic_python_agent.utils.**get_agent_params**()
Gets parameters passed to the agent via kernel cmdline or vmedia.

Parameters can be passed using either the kernel commandline or through virtual media. If boot_method is vmedia, merge params provided via vmedia with those read from the kernel command line.

Although it should never happen, if a variable is both set by vmedia and kernel command line, the setting in vmedia will take precedence.

Returns a dict of potential configuration parameters for the agent `ironic_python_agent.utils.`**`get_command_output`**(*command*)

Return the output of a given command.

Parameters **`command`** The command to be executed.

Raises CommandExecutionError if the execution of the command fails.

Returns A BytesIO string with the output.

`ironic_python_agent.utils.`**`get_efi_part_on_device`**(*device*)

Looks for the efi partition on a given device.

A boot partition on a GPT disk is assumed to be an EFI partition as well.

Parameters **`device`** lock device upon which to check for the efi partition

Returns the efi partition or None

`ironic_python_agent.utils.`**`get_journalctl_output`**(*lines=None*, *units=None*)

Query the contents of the systemd journal.

Parameters

- **`lines`** Maximum number of lines to retrieve from the logs. If None, return everything.
- **`units`** A list with the names of the units we should retrieve the logs from. If None retrieve the logs for everything.

Returns A log string.

`ironic_python_agent.utils.`**`get_node_boot_mode`**(*node*)

Returns the node boot mode.

It returns uefi if secure_boot is set to true in instance_info/capabilities of node. Otherwise it directly look for boot mode hints into

Parameters **`node`** dictionnary.

Returns bios or uefi

`ironic_python_agent.utils.`**`get_partition_table_type_from_specs`**(*node*)

Returns the node partition label, gpt or msdos.

If boot mode is uefi, return gpt. Else, choice is open, look for disk_label capabilities (instance_info has priority over properties).

Parameters **`node`**

Returns gpt or msdos

`ironic_python_agent.utils.`**`get_ssl_client_options`**(*conf*)

Format SSL-related requests options.

Parameters **`conf`** oslo_config CONF object

Returns tuple of verify and cert values to pass to requests

`ironic_python_agent.utils.`**`guess_root_disk`**(*block_devices*, *min_size_required=4294967296*)

Find suitable disk provided that root device hints are not given.

If no hints are passed, order the devices by size (primary key) and name (secondary key), and return the first device larger than min_size_required as the root disk.

`ironic_python_agent.utils.`**`gzip_and_b64encode`**(*io_dict=None*, *file_list=None*)

Gzip and base64 encode files and BytesIO buffers.

Parameters

- **`io_dict`** A dictionary containing whose the keys are the file names and the value a BytesIO object.
- **`file_list`** A list of file path.

Returns A gzipped and base64 encoded string.

`ironic_python_agent.utils.`**`is_journalctl_present`**()

Check if the journalctl command is present.

Returns True if journalctl is present, False if not.

`ironic_python_agent.utils.`**`parse_capabilities`**(*root*)

Extract capabilities from provided root dictionary-behaving object.

root.get(capabilities, {}) value can either be a dict, or a json str, or a key1:value1,key2:value2 formatted string.

Parameters **`root`** Anything behaving like a dict and containing capabilities formatted as expected. Can be node.get(properties, {}), node.get(instance_info, {}).

Returns A dictionary with the capabilities if found and well formatted, otherwise an empty dictionary.

`ironic_python_agent.utils.`**`remove_large_keys`**(*var*)

Remove specific keys from the var, recursing into dicts and lists.

`ironic_python_agent.utils.`**`scan_partition_table_type`**(*device*)

Get partition table type, msdos or gpt.

Parameters **`device_name`** the name of the device

Returns msdos, gpt or unknown

`ironic_python_agent.utils.`**`sync_clock`**(*ignore_errors=False*)

Syncs the software clock of the system.

This method syncs the system software clock if a NTP server was defined in the [DEFAULT]ntp_server configuration parameter. This method does NOT attempt to sync the hardware clock.

It will try to use either ntpdate or chrony to sync the software clock of the system. If neither is found, an exception is raised.

Parameters **`ignore_errors`** Boolean value default False that allows for the method to be called and ultimately not raise an exception. This may be useful for opportunistically attempting to sync the system software clock.

Raises CommandExecutionError if an error is encountered while attempting to sync the software clock.

`ironic_python_agent.utils.`**`try_execute`**(**cmd*, ***kwargs*)

The same as execute but returns None on error.

Executes and logs results from a system command. See docs for oslo_concurrency.processutils.execute for usage.

Instead of raising an exception on failure, this method simply returns None in case of failure.

Parameters

- **`cmd`** positional arguments to pass to processutils.execute()
- **`kwargs`** keyword arguments to pass to processutils.execute()

Raises UnknownArgumentError on receiving unknown arguments

Returns tuple of (stdout, stderr) or None in some error cases

ironic_python_agent.version module

Module contents

CHAPTER

THREE

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX