Swift Documentation

Release 2.30.2.dev8

Swift Team

Aug 03, 2024

CONTENTS

Swift is a highly available, distributed, eventually consistent object/blob store. Organizations can use Swift to store lots of data efficiently, safely, and cheaply.

This documentation is generated by the Sphinx toolkit and lives in the source tree. Additional documentation on Swift and other components of OpenStack can be found on the OpenStack wiki and at http://docs.openstack.org.

Note: If youre looking for associated projects that enhance or use Swift, please see the *Associated Projects* page.

CHAPTER

ONE

GETTING STARTED

1.1 System Requirements

Swift development currently targets Ubuntu Server 16.04, but should work on most Linux platforms.

Swift is written in Python and has these dependencies:

- Python (2.7 or 3.6-3.9)
- rsync 3.x
- liberasurecode
- The Python packages listed in the requirements file
- Testing additionally requires the test dependencies
- Testing requires these distribution packages

1.2 Development

To get started with development with Swift, or to just play around, the following docs will be useful:

- *Swift All in One* - Set up a VM with Swift installed
- *Development Guidelines*
- *First Contribution to Swift*
- *Associated Projects*

1.3 CLI client and SDK library

There are many clients in the *ecosystem*. The official CLI and SDK is python-swiftclient.

- Source code
- Python Package Index

1.4 Production

If you want to set up and configure Swift for a production cluster, the following doc should be useful:

- *Multiple Server Swift Installation*

CHAPTER

TWO

OVERVIEW AND CONCEPTS

2.1 Object Storage API overview

OpenStack Object Storage is a highly available, distributed, eventually consistent object/blob store. You create, modify, and get objects and metadata by using the Object Storage API, which is implemented as a set of Representational State Transfer (REST) web services.

For an introduction to OpenStack Object Storage, see the *OpenStack Swift Administrator Guide*.

You use the HTTPS (SSL) protocol to interact with Object Storage, and you use standard HTTP calls to perform API operations. You can also use language-specific APIs, which use the RESTful API, that make it easier for you to integrate into your applications.

To assert your right to access and change data in an account, you identify yourself to Object Storage by using an authentication token. To get a token, you present your credentials to an authentication service. The authentication service returns a token and the URL for the account. Depending on which authentication service that you use, the URL for the account appears in:

- **OpenStack Identity Service**. The URL is defined in the service catalog.
- **Tempauth**. The URL is provided in the `X-Storage-Url` response header.

In both cases, the URL is the full URL and includes the account resource.

The Object Storage API supports the standard, non-serialized response format, which is the default, and both JSON and XML serialized response formats.

The Object Storage system organizes data in a hierarchy, as follows:

- **Account**. Represents the top-level of the hierarchy.

  Your service provider creates your account and you own all resources in that account. The account defines a namespace for containers. A container might have the same name in two different accounts.

  In the OpenStack environment, *account* is synonymous with a project or tenant.

- **Container**. Defines a namespace for objects. An object with the same name in two different containers represents two different objects. You can create any number of containers within an account.

  In addition to containing objects, you can also use the container to control access to objects by using an access control list (ACL). You cannot store an ACL with individual objects.

  In addition, you configure and control many other features, such as object versioning, at the container level.

You can bulk-delete up to 10,000 containers in a single request.

You can set a storage policy on a container with predefined names and definitions from your cloud provider.

- **Object**. Stores data content, such as documents, images, and so on. You can also store custom metadata with an object.

  With the Object Storage API, you can:

  - Store an unlimited number of objects. Each object can be as large as 5 GB, which is the default. You can configure the maximum object size.
  - Upload and store objects of any size with large object creation.
  - Use cross-origin resource sharing to manage object security.
  - Compress files using content-encoding metadata.
  - Override browser behavior for an object using content-disposition metadata.
  - Schedule objects for deletion.
  - Bulk-delete up to 10,000 objects in a single request.
  - Auto-extract archive files.
  - Generate a URL that provides time-limited **GET** access to an object.
  - Upload objects directly to the Object Storage system from a browser by using form **POST** middleware.
  - Create symbolic links to other objects.

The account, container, and object hierarchy affects the way you interact with the Object Storage API.

Specifically, the resource path reflects this structure and has this format:

```
/v1/{account}/{container}/{object}
```

For example, for the `flowers/rose.jpg` object in the `images` container in the `12345678912345` account, the resource path is:

```
/v1/12345678912345/images/flowers/rose.jpg
```

Notice that the object name contains the `/` character. This slash does not indicate that Object Storage has a sub-hierarchy called `flowers` because containers do not store objects in actual sub-folders. However, the inclusion of `/` or a similar convention inside object names enables you to create pseudo-hierarchical folders and directories.

For example, if the endpoint for Object Storage is `objects.mycloud.com`, the returned URL is `https://objects.mycloud.com/v1/12345678912345`.

To access a container, append the container name to the resource path.

To access an object, append the container and the object name to the path.

If you have a large number of containers or objects, you can use query parameters to page through large lists of containers or objects. Use the `marker`, `limit`, and `end_marker` query parameters to control how many items are returned in a list and where the list starts or ends. If you want to page through in reverse order, you can use the query parameter `reverse`, noting that your marker and end_markers should be

switched when applied to a reverse listing. I.e, for a list of objects `[a, b, c, d, e]` the non-reversed could be:

```
/v1/{account}/{container}/?marker=a&end_marker=d
b
c
```

However, when reversed marker and end_marker are applied to a reversed list:

```
/v1/{account}/{container}/?marker=d&end_marker=a&reverse=on
c
b
```

Object Storage HTTP requests have the following default constraints. Your service provider might use different default values.

Item	Maximum value	Notes
Number of HTTP headers	90	
Length of HTTP headers	4096 bytes	
Length per HTTP request line	8192 bytes	
Length of HTTP request	5 GB	
Length of container names	256 bytes	Cannot contain the / character.
Length of object names	1024 bytes	By default, there are no character restrictions.

You must UTF-8-encode and then URL-encode container and object names before you call the API binding. If you use an API binding that performs the URL-encoding for you, do not URL-encode the names before you call the API binding. Otherwise, you double-encode these names. Check the length restrictions against the URL-encoded string.

The API Reference describes the operations that you can perform with the Object Storage API:

- Storage accounts: Use to perform account-level tasks.

  Lists containers for a specified account. Creates, updates, and deletes account metadata. Shows account metadata.

- Storage containers: Use to perform container-level tasks.

  Lists objects in a specified container. Creates, shows details for, and deletes containers. Creates, updates, shows, and deletes container metadata.

- Storage objects: Use to perform object-level tasks.

  Creates, replaces, shows details for, and deletes objects. Copies objects with another object with a new or different name. Updates object metadata.

2.2 Swift Architectural Overview

2.2.1 Proxy Server

The Proxy Server is responsible for tying together the rest of the Swift architecture. For each request, it will look up the location of the account, container, or object in the ring (see below) and route the request accordingly. For Erasure Code type policies, the Proxy Server is also responsible for encoding and decoding object data. See *Erasure Code Support* for complete information on Erasure Code support. The public API is also exposed through the Proxy Server.

A large number of failures are also handled in the Proxy Server. For example, if a server is unavailable for an object PUT, it will ask the ring for a handoff server and route there instead.

When objects are streamed to or from an object server, they are streamed directly through the proxy server to or from the user the proxy server does not spool them.

2.2.2 The Ring

A ring represents a mapping between the names of entities stored on disk and their physical location. There are separate rings for accounts, containers, and one object ring per storage policy. When other components need to perform any operation on an object, container, or account, they need to interact with the appropriate ring to determine its location in the cluster.

The Ring maintains this mapping using zones, devices, partitions, and replicas. Each partition in the ring is replicated, by default, 3 times across the cluster, and the locations for a partition are stored in the mapping maintained by the ring. The ring is also responsible for determining which devices are used for handoff in failure scenarios.

The replicas of each partition will be isolated onto as many distinct regions, zones, servers and devices as the capacity of these failure domains allow. If there are less failure domains at a given tier than replicas of the partition assigned within a tier (e.g. a 3 replica cluster with 2 servers), or the available capacity across the failure domains within a tier are not well balanced it will not be possible to achieve both even capacity distribution (*balance*) as well as complete isolation of replicas across failure domains (*dispersion*). When this occurs the ring management tools will display a warning so that the operator can evaluate the cluster topology.

Data is evenly distributed across the capacity available in the cluster as described by the devices weight. Weights can be used to balance the distribution of partitions on drives across the cluster. This can be useful, for example, when different sized drives are used in a cluster. Device weights can also be used when adding or removing capacity or failure domains to control how many partitions are reassigned during a rebalance to be moved as soon as replication bandwidth allows.

Note: Prior to Swift 2.1.0 it was not possible to restrict partition movement by device weight when adding new failure domains, and would allow extremely unbalanced rings. The greedy dispersion algorithm is now subject to the constraints of the physical capacity in the system, but can be adjusted with-in reason via the overload option. Artificially unbalancing the partition assignment without respect to capacity can introduce unexpected full devices when a given failure domain does not physically support its share of the used capacity in the tier.

When partitions need to be moved around (for example if a device is added to the cluster), the ring ensures that a minimum number of partitions are moved at a time, and only one replica of a partition is moved at

a time.

The ring is used by the Proxy server and several background processes (like replication). See *The Rings* for complete information on the ring.

2.2.3 Storage Policies

Storage Policies provide a way for object storage providers to differentiate service levels, features and behaviors of a Swift deployment. Each Storage Policy configured in Swift is exposed to the client via an abstract name. Each device in the system is assigned to one or more Storage Policies. This is accomplished through the use of multiple object rings, where each Storage Policy has an independent object ring, which may include a subset of hardware implementing a particular differentiation.

For example, one might have the default policy with 3x replication, and create a second policy which, when applied to new containers only uses 2x replication. Another might add SSDs to a set of storage nodes and create a performance tier storage policy for certain containers to have their objects stored there. Yet another might be the use of Erasure Coding to define a cold-storage tier.

This mapping is then exposed on a per-container basis, where each container can be assigned a specific storage policy when it is created, which remains in effect for the lifetime of the container. Applications require minimal awareness of storage policies to use them; once a container has been created with a specific policy, all objects stored in it will be done so in accordance with that policy.

The Storage Policies feature is implemented throughout the entire code base so it is an important concept in understanding Swift architecture.

See *Storage Policies* for complete information on storage policies.

2.2.4 Object Server

The Object Server is a very simple blob storage server that can store, retrieve and delete objects stored on local devices. Objects are stored as binary files on the filesystem with metadata stored in the files extended attributes (xattrs). This requires that the underlying filesystem choice for object servers support xattrs on files. Some filesystems, like ext3, have xattrs turned off by default.

Each object is stored using a path derived from the object names hash and the operations timestamp. Last write always wins, and ensures that the latest object version will be served. A deletion is also treated as a version of the file (a 0 byte file ending with .ts, which stands for tombstone). This ensures that deleted files are replicated correctly and older versions dont magically reappear due to failure scenarios.

2.2.5 Container Server

The Container Servers primary job is to handle listings of objects. It doesnt know where those objects are, just what objects are in a specific container. The listings are stored as sqlite database files, and replicated across the cluster similar to how objects are. Statistics are also tracked that include the total number of objects, and total storage usage for that container.

2.2.6 Account Server

The Account Server is very similar to the Container Server, excepting that it is responsible for listings of containers rather than objects.

2.2.7 Replication

Replication is designed to keep the system in a consistent state in the face of temporary error conditions like network outages or drive failures.

The replication processes compare local data with each remote copy to ensure they all contain the latest version. Object replication uses a hash list to quickly compare subsections of each partition, and container and account replication use a combination of hashes and shared high water marks.

Replication updates are push based. For object replication, updating is just a matter of rsyncing files to the peer. Account and container replication push missing records over HTTP or rsync whole database files.

The replicator also ensures that data is removed from the system. When an item (object, container, or account) is deleted, a tombstone is set as the latest version of the item. The replicator will see the tombstone and ensure that the item is removed from the entire system.

See *Replication* for complete information on replication.

2.2.8 Reconstruction

The reconstructor is used by Erasure Code policies and is analogous to the replicator for Replication type policies. See *Erasure Code Support* for complete information on both Erasure Code support as well as the reconstructor.

2.2.9 Updaters

There are times when container or account data can not be immediately updated. This usually occurs during failure scenarios or periods of high load. If an update fails, the update is queued locally on the filesystem, and the updater will process the failed updates. This is where an eventual consistency window will most likely come in to play. For example, suppose a container server is under load and a new object is put in to the system. The object will be immediately available for reads as soon as the proxy server responds to the client with success. However, the container server did not update the object listing, and so the update would be queued for a later update. Container listings, therefore, may not immediately contain the object.

In practice, the consistency window is only as large as the frequency at which the updater runs and may not even be noticed as the proxy server will route listing requests to the first container server which responds. The server under load may not be the one that serves subsequent listing requests one of the other two replicas may handle the listing.

2.2.10 Auditors

Auditors crawl the local server checking the integrity of the objects, containers, and accounts. If corruption is found (in the case of bit rot, for example), the file is quarantined, and replication will replace the bad file from another replica. If other errors are found they are logged (for example, an objects listing cant be found on any container server it should be).

2.3 The Rings

The rings determine where data should reside in the cluster. There is a separate ring for account databases, container databases, and individual object storage policies but each ring works in the same way. These rings are externally managed. The server processes themselves do not modify the rings; they are instead given new rings modified by other tools.

The ring uses a configurable number of bits from the MD5 hash of an items path as a partition index that designates the device(s) on which that item should be stored. The number of bits kept from the hash is known as the partition power, and 2 to the partition power indicates the partition count. Partitioning the full MD5 hash ring allows the cluster components to process resources in batches. This ends up either more efficient or at least less complex than working with each item separately or the entire cluster all at once.

Another configurable value is the replica count, which indicates how many devices to assign for each partition in the ring. By having multiple devices responsible for each partition, the cluster can recover from drive or network failures.

Devices are added to the ring to describe the capacity available for partition replica assignments. Devices are placed into failure domains consisting of region, zone, and server. Regions can be used to describe geographical systems characterized by lower bandwidth or higher latency between machines in different regions. Many rings will consist of only a single region. Zones can be used to group devices based on physical locations, power separations, network separations, or any other attribute that would lessen multiple replicas being unavailable at the same time.

Devices are given a weight which describes the relative storage capacity contributed by the device in comparison to other devices.

When building a ring, replicas for each partition will be assigned to devices according to the devices weights. Additionally, each replica of a partition will preferentially be assigned to a device whose failure domain does not already have a replica for that partition. Only a single replica of a partition may be assigned to each device - you must have at least as many devices as replicas.

2.3.1 Ring Builder

The rings are built and managed manually by a utility called the ring-builder. The ring-builder assigns partitions to devices and writes an optimized structure to a gzipped, serialized file on disk for shipping out to the servers. The server processes check the modification time of the file occasionally and reload their in-memory copies of the ring structure as needed. Because of how the ring-builder manages changes to the ring, using a slightly older ring usually just means that for a subset of the partitions the device for one of the replicas will be incorrect, which can be easily worked around.

The ring-builder also keeps a separate builder file which includes the ring information as well as additional data required to build future rings. It is very important to keep multiple backup copies of these

builder files. One option is to copy the builder files out to every server while copying the ring files themselves. Another is to upload the builder files into the cluster itself. Complete loss of a builder file will mean creating a new ring from scratch, nearly all partitions will end up assigned to different devices, and therefore nearly all data stored will have to be replicated to new locations. So, recovery from a builder file loss is possible, but data will definitely be unreachable for an extended time.

2.3.2 Ring Data Structure

The ring data structure consists of three top level fields: a list of devices in the cluster, a list of lists of device ids indicating partition to device assignments, and an integer indicating the number of bits to shift an MD5 hash to calculate the partition for the hash.

List of Devices

The list of devices is known internally to the Ring class as `devs`. Each item in the list of devices is a dictionary with the following keys:

id	integer	The index into the list of devices.
zone	integer	The zone in which the device resides.
region	integer	The region in which the zone resides.
weight	float	The relative weight of the device in comparison to other devices. This usually corresponds directly to the amount of disk space the device has compared to other devices. For instance a device with 1 terabyte of space might have a weight of 100.0 and another device with 2 terabytes of space might have a weight of 200.0. This weight can also be used to bring back into balance a device that has ended up with more or less data than desired over time. A good average weight of 100.0 allows flexibility in lowering the weight later if necessary.
ip	string	The IP address or hostname of the server containing the device.
port	int	The TCP port on which the server process listens to serve requests for the device.
device	string	The on-disk name of the device on the server. For example: `sdb1`
meta	string	A general-use field for storing additional information for the device. This information isnt used directly by the server processes, but can be useful in debugging. For example, the date and time of installation and hardware manufacturer could be stored here.

Note: The list of devices may contain holes, or indexes set to `None`, for devices that have been removed from the cluster. However, device ids are reused. Device ids are reused to avoid potentially running out of device id slots when there are available slots (from prior removal of devices). A consequence of this device id reuse is that the device id (integer value) does not necessarily correspond with the chronology of when the device was added to the ring. Also, some devices may be temporarily disabled by setting their weight to `0.0`. To obtain a list of active devices (for uptime polling, for example) the Python code would look like:

```
devices = list(self._iter_devs())
```

Partition Assignment List

The partition assignment list is known internally to the Ring class as `_replica2part2dev_id`. This is a list of `array('H')`s, one for each replica. Each `array('H')` has a length equal to the partition count for the ring. Each integer in the `array('H')` is an index into the above list of devices.

So, to create a list of device dictionaries assigned to a partition, the Python code would look like:

```
devices = [self.devs[part2dev_id[partition]]
           for part2dev_id in self._replica2part2dev_id]
```

`array('H')` is used for memory conservation as there may be millions of partitions.

Partition Shift Value

The partition shift value is known internally to the Ring class as `_part_shift`. This value is used to shift an MD5 hash of an items path to calculate the partition on which the data for that item should reside. Only the top four bytes of the hash are used in this process. For example, to compute the partition for the path `/account/container/object`, the Python code might look like:

```
objhash = md5('/account/container/object').digest()
partition = struct.unpack_from('>I', objhash)[0] >> self._part_shift
```

For a ring generated with partition power `P`, the partition shift value is `32 - P`.

Fractional Replicas

A ring is not restricted to having an integer number of replicas. In order to support the gradual changing of replica counts, the ring is able to have a real number of replicas.

When the number of replicas is not an integer, the last element of `_replica2part2dev_id` will have a length that is less than the partition count for the ring. This means that some partitions will have more replicas than others. For example, if a ring has `3.25` replicas, then 25% of its partitions will have four replicas, while the remaining 75% will have just three.

Dispersion

With each rebalance, the ring builder calculates a dispersion metric. This is the percentage of partitions in the ring that have too many replicas within a particular failure domain.

For example, if you have three servers in a cluster but two replicas for a partition get placed onto the same server, that partition will count towards the dispersion metric.

A lower dispersion value is better, and the value can be used to find the proper value for overload.

Overload

The ring builder tries to keep replicas as far apart as possible while still respecting device weights. When it cant do both, the overload factor determines what happens. Each device may take some extra fraction of its desired partitions to allow for replica dispersion; once that extra fraction is exhausted, replicas will be placed closer together than is optimal for durability.

Essentially, the overload factor lets the operator trade off replica dispersion (durability) against device balance (uniform disk usage).

The default overload factor is `0`, so device weights will be strictly followed.

With an overload factor of `0.1`, each device will accept 10% more partitions than it otherwise would, but only if needed to maintain dispersion.

Example: Consider a 3-node cluster of machines with equal-size disks; let node A have 12 disks, node B have 12 disks, and node C have only 11 disks. Let the ring have an overload factor of `0.1` (10%).

Without the overload, some partitions would end up with replicas only on nodes A and B. However, with the overload, every device is willing to accept up to 10% more partitions for the sake of dispersion. The missing disk in C means there is one disks worth of partitions that would like to spread across the remaining 11 disks, which gives each disk in C an extra 9.09% load. Since this is less than the 10% overload, there is one replica of each partition on each node.

However, this does mean that the disks in node C will have more data on them than the disks in nodes A and B. If 80% full is the warning threshold for the cluster, node Cs disks will reach 80% full while A and Bs disks are only 72.7% full.

2.3.3 Partition & Replica Terminology

All descriptions of consistent hashing describe the process of breaking the keyspace up into multiple ranges (vnodes, buckets, etc.) - many more than the number of nodes to which keys in the keyspace must be assigned. Swift calls these ranges *partitions* - they are partitions of the total keyspace.

Each partition will have multiple replicas. Every replica of each partition must be assigned to a device in the ring. When describing a specific replica of a partition (like when its assigned a device) it is described as a *part-replica* in that it is a specific *replica* of the specific *partition*. A single device will likely be assigned different replicas from many partitions, but it may not be assigned multiple replicas of a single partition.

The total number of partitions in a ring is calculated as `2 ** <part-power>`. The total number of part-replicas in a ring is calculated as `<replica-count> * 2 ** <part-power>`.

When considering a devices *weight* it is useful to describe the number of part-replicas it would like to be assigned. A single device, regardless of weight, will never hold more than `2 ** <part-power>` part-replicas because it can not have more than one replica of any partition assigned. The number of part-replicas a device can take by weights is calculated as its *parts-wanted*. The true number of part-replicas assigned to a device can be compared to its parts-wanted similarly to a calculation of percentage error - this deviation in the observed result from the idealized target is called a devices *balance*.

When considering a devices *failure domain* it is useful to describe the number of part-replicas it would like to be assigned. The number of part-replicas wanted in a failure domain of a tier is the sum of the part-replicas wanted in the failure domains of its sub-tier. However, collectively when the total number of part-replicas in a failure domain exceeds or is equal to `2 ** <part-power>` it is most obvious that its no longer sufficient to consider only the number of total part-replicas, but rather the fraction of each replicas partitions. Consider for example a ring with 3 replicas and 3 servers: while dispersion requires

that each server hold only of the total part-replicas, placement is additionally constrained to require `1.0` replica of *each* partition per server. It would not be sufficient to satisfy dispersion if two devices on one of the servers each held a replica of a single partition, while another server held none. By considering a decimal fraction of one replicas worth of partitions in a failure domain we can derive the total part-replicas wanted in a failure domain (`1.0 * 2 ** <part-power>`). Additionally we infer more about *which* part-replicas must go in the failure domain. Consider a ring with three replicas and two zones, each with two servers (four servers total). The three replicas worth of partitions will be assigned into two failure domains at the zone tier. Each zone must hold more than one replica of some partitions. We represent this improper fraction of a replicas worth of partitions in decimal form as `1.5` (`3.0 / 2`). This tells us not only the *number* of total partitions (`1.5 * 2 ** <part-power>`) but also that *each* partition must have *at least* one replica in this failure domain (in fact `0.5` of the partitions will have 2 replicas). Within each zone the two servers will hold `0.75` of a replicas worth of partitions - this is equal both to the fraction of a replicas worth of partitions assigned to each zone (`1.5`) divided evenly among the number of failure domains in its sub-tier (2 servers in each zone, i.e. `1.5 / 2`) but *also* the total number of replicas (`3.0`) divided evenly among the total number of failure domains in the server tier (2 servers Œ 2 zones = 4, i.e. `3.0 / 4`). It is useful to consider that each server in this ring will hold only `0.75` of a replicas worth of partitions which tells that any server should have *at most* one replica of a given partition assigned. In the interests of brevity, some variable names will often refer to the concept representing the fraction of a replicas worth of partitions in decimal form as *replicanths* - this is meant to invoke connotations similar to ordinal numbers as applied to fractions, but generalized to a replica instead of a four*th* or a fif*th*. The n was probably thrown in because of Blade Runner.

2.3.4 Building the Ring

First the ring builder calculates the replicanths wanted at each tier in the rings topology based on weight.

Then the ring builder calculates the replicanths wanted at each tier in the rings topology based on dispersion.

Then the ring builder calculates the maximum deviation on a single device between its weighted replicanths and wanted replicanths.

Next we interpolate between the two replicanth values (weighted & wanted) at each tier using the specified overload (up to the maximum required overload). Its a linear interpolation, similar to solving for a point on a line between two points - we calculate the slope across the max required overload and then calculate the intersection of the line with the desired overload. This becomes the target.

From the target we calculate the minimum and maximum number of replicas any partition may have in a tier. This becomes the *replica-plan*.

Finally, we calculate the number of partitions that should ideally be assigned to each device based the replica-plan.

On initial balance (i.e., the first time partitions are placed to generate a ring) we must assign each replica of each partition to the device that desires the most partitions excluding any devices that already have their maximum number of replicas of that partition assigned to some parent tier of that devices failure domain.

When building a new ring based on an old ring, the desired number of partitions each device wants is recalculated from the current replica-plan. Next the partitions to be reassigned are gathered up. Any removed devices have all their assigned partitions unassigned and added to the gathered list. Any partition replicas that (due to the addition of new devices) can be spread out for better durability are unassigned and added to the gathered list. Any devices that have more partitions than they now desire have random

partitions unassigned from them and added to the gathered list. Lastly, the gathered partitions are then reassigned to devices using a similar method as in the initial assignment described above.

Whenever a partition has a replica reassigned, the time of the reassignment is recorded. This is taken into account when gathering partitions to reassign so that no partition is moved twice in a configurable amount of time. This configurable amount of time is known internally to the RingBuilder class as `min_part_hours`. This restriction is ignored for replicas of partitions on devices that have been removed, as device removal should only happens on device failure and theres no choice but to make a reassignment.

The above processes dont always perfectly rebalance a ring due to the random nature of gathering partitions for reassignment. To help reach a more balanced ring, the rebalance process is repeated a fixed number of times until the replica-plan is fulfilled or unable to be fulfilled (indicating we probably cant get perfect balance due to too many partitions recently moved).

2.3.5 Composite Rings

See *Composite Ring Builder*.

swift-ring-composer (Experimental)

`swift-ring-composer` is an experimental tool for building a composite ring file from other existing component ring builder files. Its CLI, name or implementation may change or be removed altogether in future versions of Swift.

Currently its interface is similar to that of the `swift-ring-builder`. The command structure takes the form of:

```
swift-ring-composer <composite builder file> <sub-command> <options>
```

where `<composite builder file>` is a special builder which stores a json blob of composite ring metadata. This metadata describes the component `RingBuilder`s used in the composite ring, their order and version.

There are currently 2 sub-commands: `show` and `compose`. The `show` sub-command takes no additional arguments and displays the current contents of of the composite builder file:

```
swift-ring-composer <composite builder file> show
```

The `compose` sub-command is the one that actually stitches the component ring builders together to create both the composite ring file and composite builder file. The command takes the form:

```
swift-ring-composer <composite builder file> compose <builder1> \
<builder2> [<builder3> .. <builderN>] --output <composite ring file> \
[--force]
```

There may look like there is a lot going on there but its actually quite simple. The `compose` command takes in the list of builders to stitch together and the filename for the composite ring file via the `--output` option. The `--force` option overrides checks on the ring composition.

To change ring devices, first add or remove devices from the component ring builders and then use the `compose` sub-command to create a new composite ring file.

Note: `swift-ring-builder` cannot be used to inspect the generated composite ring file because there is no conventional builder file corresponding to the composite ring file name. You can either programmatically look inside the composite ring file using the swift ring classes or create a temporary builder file from the composite ring file using:

```
swift-ring-builder <composite ring file> write_builder
```

Do not use this builder file to manage ring devices.

For further details use:

```
swift-ring-composer -h
```

2.3.6 Ring Builder Analyzer

This is a tool for analyzing how well the ring builder performs its job in a particular scenario. It is intended to help developers quantify any improvements or regressions in the ring builder; it is probably not useful to others.

The ring builder analyzer takes a scenario file containing some initial parameters for a ring builder plus a certain number of rounds. In each round, some modifications are made to the builder, e.g. add a device, remove a device, change a devices weight. Then, the builder is repeatedly rebalanced until it settles down. Data about that round is printed, and the next round begins.

Scenarios are specified in JSON. Example scenario for a gradual device addition:

```
{
    "part_power": 12,
    "replicas": 3,
    "overload": 0.1,
    "random_seed": 203488,

    "rounds": [
        [
            ["add", "r1z2-10.20.30.40:6200/sda", 8000],
            ["add", "r1z2-10.20.30.40:6200/sdb", 8000],
            ["add", "r1z2-10.20.30.40:6200/sdc", 8000],
            ["add", "r1z2-10.20.30.40:6200/sdd", 8000],

            ["add", "r1z2-10.20.30.41:6200/sda", 8000],
            ["add", "r1z2-10.20.30.41:6200/sdb", 8000],
            ["add", "r1z2-10.20.30.41:6200/sdc", 8000],
            ["add", "r1z2-10.20.30.41:6200/sdd", 8000],

            ["add", "r1z2-10.20.30.43:6200/sda", 8000],
            ["add", "r1z2-10.20.30.43:6200/sdb", 8000],
            ["add", "r1z2-10.20.30.43:6200/sdc", 8000],
            ["add", "r1z2-10.20.30.43:6200/sdd", 8000],

            ["add", "r1z2-10.20.30.44:6200/sda", 8000],
```

(continues on next page)

(continued from previous page)

```
            ["add", "r1z2-10.20.30.44:6200/sdb", 8000],
            ["add", "r1z2-10.20.30.44:6200/sdc", 8000]
        ], [
            ["add", "r1z2-10.20.30.44:6200/sdd", 1000]
        ], [
            ["set_weight", 15, 2000]
        ], [
            ["remove", 3],
            ["set_weight", 15, 3000]
        ], [
            ["set_weight", 15, 4000]
        ], [
            ["set_weight", 15, 5000]
        ], [
            ["set_weight", 15, 6000]
        ], [
            ["set_weight", 15, 7000]
        ], [
            ["set_weight", 15, 8000]
        ]]
}
```

2.3.7 History

The ring code went through many iterations before arriving at what it is now and while it has largely been stable, the algorithm has seen a few tweaks or perhaps even fundamentally changed as new ideas emerge. This section will try to describe the previous ideas attempted and attempt to explain why they were discarded.

A live ring option was considered where each server could maintain its own copy of the ring and the servers would use a gossip protocol to communicate the changes they made. This was discarded as too complex and error prone to code correctly in the project timespan available. One bug could easily gossip bad data out to the entire cluster and be difficult to recover from. Having an externally managed ring simplifies the process, allows full validation of data before its shipped out to the servers, and guarantees each server is using a ring from the same timeline. It also means that the servers themselves arent spending a lot of resources maintaining rings.

A couple of ring server options were considered. One was where all ring lookups would be done by calling a service on a separate server or set of servers, but this was discarded due to the latency involved. Another was much like the current process but where servers could submit change requests to the ring server to have a new ring built and shipped back out to the servers. This was discarded due to project time constraints and because ring changes are currently infrequent enough that manual control was sufficient. However, lack of quick automatic ring changes did mean that other components of the system had to be coded to handle devices being unavailable for a period of hours until someone could manually update the ring.

The current ring process has each replica of a partition independently assigned to a device. A version of the ring that used a third of the memory was tried, where the first replica of a partition was directly assigned and the other two were determined by walking the ring until finding additional devices in other zones. This was discarded due to the loss of control over how many replicas for a given partition moved

at once. Keeping each replica independent allows for moving only one partition replica within a given time window (except due to device failures). Using the additional memory was deemed a good trade-off for moving data around the cluster much less often.

Another ring design was tried where the partition to device assignments werent stored in a big list in memory but instead each device was assigned a set of hashes, or anchors. The partition would be determined from the data items hash and the nearest device anchors would determine where the replicas should be stored. However, to get reasonable distribution of data each device had to have a lot of anchors and walking through those anchors to find replicas started to add up. In the end, the memory savings wasnt that great and more processing power was used, so the idea was discarded.

A completely non-partitioned ring was also tried but discarded as the partitioning helps many other components of the system, especially replication. Replication can be attempted and retried in a partition batch with the other replicas rather than each data item independently attempted and retried. Hashes of directory structures can be calculated and compared with other replicas to reduce directory walking and network traffic.

Partitioning and independently assigning partition replicas also allowed for the best-balanced cluster. The best of the other strategies tended to give ś10% variance on device balance with devices of equal weight and ś15% with devices of varying weights. The current strategy allows us to get ś3% and ś8% respectively.

Various hashing algorithms were tried. SHA offers better security, but the ring doesnt need to be cryptographically secure and SHA is slower. Murmur was much faster, but MD5 was built-in and hash computation is a small percentage of the overall request handling time. In all, once it was decided the servers wouldnt be maintaining the rings themselves anyway and only doing hash lookups, MD5 was chosen for its general availability, good distribution, and adequate speed.

The placement algorithm has seen a number of behavioral changes for unbalanceable rings. The ring builder wants to keep replicas as far apart as possible while still respecting device weights. In most cases, the ring builder can achieve both, but sometimes they conflict. At first, the behavior was to keep the replicas far apart and ignore device weight, but that made it impossible to gradually go from one region to two, or from two to three. Then it was changed to favor device weight over dispersion, but that wasnt so good for rings that were close to balanceable, like 3 machines with 60TB, 60TB, and 57TB of disk space; operators were expecting one replica per machine, but didnt always get it. After that, overload was added to the ring builder so that operators could choose a balance between dispersion and device weights. In time the overload concept was improved and made more accurate.

For more background on consistent hashing rings, please see *Building a Consistent Hashing Ring*.

2.4 Storage Policies

Storage Policies allow for some level of segmenting the cluster for various purposes through the creation of multiple object rings. The Storage Policies feature is implemented throughout the entire code base so it is an important concept in understanding Swift architecture.

As described in *The Rings*, Swift uses modified hashing rings to determine where data should reside in the cluster. There is a separate ring for account databases, container databases, and there is also one object ring per storage policy. Each object ring behaves exactly the same way and is maintained in the same manner, but with policies, different devices can belong to different rings. By supporting multiple object rings, Swift allows the application and/or deployer to essentially segregate the object storage within a single cluster. There are many reasons why this might be desirable:

- Different levels of durability: If a provider wants to offer, for example, 2x replication and 3x replication but doesnt want to maintain 2 separate clusters, they would setup a 2x and a 3x replication policy and assign the nodes to their respective rings. Furthermore, if a provider wanted to offer a cold storage tier, they could create an erasure coded policy.
- Performance: Just as SSDs can be used as the exclusive members of an account or database ring, an SSD-only object ring can be created as well and used to implement a low-latency/high performance policy.
- Collecting nodes into group: Different object rings may have different physical servers so that objects in specific storage policies are always placed in a particular data center or geography.
- Different Storage implementations: Another example would be to collect together a set of nodes that use a different Diskfile (e.g., Kinetic, GlusterFS) and use a policy to direct traffic just to those nodes.
- Different read and write affinity settings: proxy-servers can be configured to use different read and write affinity options for each policy. See *Per policy configuration* for more details.

Note: Today, Swift supports two different policy types: Replication and Erasure Code. See *Erasure Code Support* for details.

Also note that Diskfile refers to backend object storage plug-in architecture. See *Pluggable On-Disk Back-end APIs* for details.

2.4.1 Containers and Policies

Policies are implemented at the container level. There are many advantages to this approach, not the least of which is how easy it makes life on applications that want to take advantage of them. It also ensures that Storage Policies remain a core feature of Swift independent of the auth implementation. Policies were not implemented at the account/auth layer because it would require changes to all auth systems in use by Swift deployers. Each container has a new special immutable metadata element called the storage policy index. Note that internally, Swift relies on policy indexes and not policy names. Policy names exist for human readability and translation is managed in the proxy. When a container is created, one new optional header is supported to specify the policy name. If no name is specified, the default policy is used (and if no other policies defined, Policy-0 is considered the default). We will be covering the difference between default and Policy-0 in the next section.

Policies are assigned when a container is created. Once a container has been assigned a policy, it cannot be changed (unless it is deleted/recreated). The implications on data placement/movement for large datasets would make this a task best left for applications to perform. Therefore, if a container has an existing policy of, for example 3x replication, and one wanted to migrate that data to an Erasure Code policy, the application would create another container specifying the other policy parameters and then simply move the data from one container to the other. Policies apply on a per container basis allowing for minimal application awareness; once a container has been created with a specific policy, all objects stored in it will be done so in accordance with that policy. If a container with a specific name is deleted (requires the container be empty) a new container may be created with the same name without any restriction on storage policy enforced by the deleted container which previously shared the same name.

Containers have a many-to-one relationship with policies meaning that any number of containers can share one policy. There is no limit to how many containers can use a specific policy.

The notion of associating a ring with a container introduces an interesting scenario: What would happen if 2 containers of the same name were created with different Storage Policies on either side of a network outage at the same time? Furthermore, what would happen if objects were placed in those containers, a whole bunch of them, and then later the network outage was restored? Well, without special care it would be a big problem as an application could end up using the wrong ring to try and find an object. Luckily there is a solution for this problem, a daemon known as the Container Reconciler works tirelessly to identify and rectify this potential scenario.

2.4.2 Container Reconciler

Because atomicity of container creation cannot be enforced in a distributed eventually consistent system, object writes into the wrong storage policy must be eventually merged into the correct storage policy by an asynchronous daemon. Recovery from object writes during a network partition which resulted in a split brain container created with different storage policies are handled by the *swift-container-reconciler* daemon.

The container reconciler works off a queue similar to the object-expirer. The queue is populated during container-replication. It is never considered incorrect to enqueue an object to be evaluated by the container-reconciler because if there is nothing wrong with the location of the object the reconciler will simply dequeue it. The container-reconciler queue is an indexed log for the real location of an object for which a discrepancy in the storage policy of the container was discovered.

To determine the correct storage policy of a container, it is necessary to update the status_changed_at field in the container_stat table when a container changes status from deleted to re-created. This transaction log allows the container-replicator to update the correct storage policy both when replicating a container and handling REPLICATE requests.

Because each object write is a separate distributed transaction it is not possible to determine the correctness of the storage policy for each object write with respect to the entire transaction log at a given container database. As such, container databases will always record the object write regardless of the storage policy on a per object row basis. Object byte and count stats are tracked per storage policy in each container and reconciled using normal object row merge semantics.

The object rows are ensured to be fully durable during replication using the normal container replication. After the container replicator pushes its object rows to available primary nodes any misplaced object rows are bulk loaded into containers based off the object timestamp under the `.misplaced_objects` system account. The rows are initially written to a handoff container on the local node, and at the end of the replication pass the `.misplaced_objects` containers are replicated to the correct primary nodes.

The container-reconciler processes the `.misplaced_objects` containers in descending order and reaps its containers as the objects represented by the rows are successfully reconciled. The container-reconciler will always validate the correct storage policy for enqueued objects using direct container HEAD requests which are accelerated via caching.

Because failure of individual storage nodes in aggregate is assumed to be common at scale, the container-reconciler will make forward progress with a simple quorum majority. During a combination of failures and rebalances it is possible that a quorum could provide an incomplete record of the correct storage policy - so an object write may have to be applied more than once. Because storage nodes and container databases will not process writes with an `X-Timestamp` less than or equal to their existing record when objects writes are re-applied their timestamp is slightly incremented. In order for this increment to be applied transparently to the client a second vector of time has been added to Swift for internal use. See *Timestamp*.

As the reconciler applies object writes to the correct storage policy it cleans up writes which no longer

apply to the incorrect storage policy and removes the rows from the `.misplaced_objects` containers. After all rows have been successfully processed it sleeps and will periodically check for newly enqueued rows to be discovered during container replication.

2.4.3 Default versus Policy-0

Storage Policies is a versatile feature intended to support both new and pre-existing clusters with the same level of flexibility. For that reason, we introduce the `Policy-0` concept which is not the same as the default policy. As you will see when we begin to configure policies, each policy has a single name and an arbitrary number of aliases (human friendly, configurable) as well as an index (or simply policy number). Swift reserves index 0 to map to the object ring thats present in all installations (e.g., `/etc/swift/object.ring.gz`). You can name this policy anything you like, and if no policies are defined it will report itself as `Policy-0`, however you cannot change the index as there must always be a policy with index 0.

Another important concept is the default policy which can be any policy in the cluster. The default policy is the policy that is automatically chosen when a container creation request is sent without a storage policy being specified. *Configuring Policies* describes how to set the default policy. The difference from `Policy-0` is subtle but extremely important. `Policy-0` is what is used by Swift when accessing pre-storage-policy containers which wont have a policy - in this case we would not use the default as it might not have the same policy as legacy containers. When no other policies are defined, Swift will always choose `Policy-0` as the default.

In other words, default means create using this policy if nothing else is specified and `Policy-0` means use the legacy policy if a container doesnt have one which really means use `object.ring.gz` for lookups.

Note: With the Storage Policy based code, its not possible to create a container that doesnt have a policy. If nothing is provided, Swift will still select the default and assign it to the container. For containers created before Storage Policies were introduced, the legacy Policy-0 will be used.

2.4.4 Deprecating Policies

There will be times when a policy is no longer desired; however simply deleting the policy and associated rings would be problematic for existing data. In order to ensure that resources are not orphaned in the cluster (left on disk but no longer accessible) and to provide proper messaging to applications when a policy needs to be retired, the notion of deprecation is used. *Configuring Policies* describes how to deprecate a policy.

Swifts behavior with deprecated policies is as follows:

- The deprecated policy will not appear in /info
- PUT/GET/DELETE/POST/HEAD are still allowed on the pre-existing containers created with a deprecated policy
- Clients will get an 400 Bad Request error when trying to create a new container using the deprecated policy
- Clients still have access to policy statistics via HEAD on pre-existing containers

Note: A policy cannot be both the default and deprecated. If you deprecate the default policy, you must specify a new default.

You can also use the deprecated feature to rollout new policies. If you want to test a new storage policy before making it generally available you could deprecate the policy when you initially roll it the new configuration and rings to all nodes. Being deprecated will render it innate and unable to be used. To test it you will need to create a container with that storage policy; which will require a single proxy instance (or a set of proxy-servers which are only internally accessible) that has been one-off configured with the new policy NOT marked deprecated. Once the container has been created with the new storage policy any client authorized to use that container will be able to add and access data stored in that container in the new storage policy. When satisfied you can roll out a new `swift.conf` which does not mark the policy as deprecated to all nodes.

2.4.5 Configuring Policies

Note: See *Adding Storage Policies to an Existing SAIO* for a step by step guide on adding a policy to the SAIO setup.

It is important that the deployer have a solid understanding of the semantics for configuring policies. Configuring a policy is a three-step process:

1. Edit your `/etc/swift/swift.conf` file to define your new policy.
2. Create the corresponding policy object ring file.
3. (Optional) Create policy-specific proxy-server configuration settings.

Defining a policy

Each policy is defined by a section in the `/etc/swift/swift.conf` file. The section name must be of the form `[storage-policy:<N>]` where `<N>` is the policy index. Theres no reason other than readability that policy indexes be sequential but the following rules are enforced:

- If a policy with index `0` is not declared and no other policies are defined, Swift will create a default policy with index `0`.
- The policy index must be a non-negative integer.
- Policy indexes must be unique.

Warning: The index of a policy should never be changed once a policy has been created and used. Changing a policy index may cause loss of access to data.

Each policy section contains the following options:

- **`name = <policy_name>` (required)**
  - The primary name of the policy.
  - Policy names are case insensitive.

  - Policy names must contain only letters, digits or a dash.
  - Policy names must be unique.
  - Policy names can be changed.
  - The name `Policy-0` can only be used for the policy with index `0`.
  - To avoid confusion with policy indexes it is strongly recommended that policy names are not numbers (e.g. 1). However, for backwards compatibility, names that are numbers are supported.

- **`aliases = <policy_name>[, <policy_name>, ...]` (optional)**
  - A comma-separated list of alternative names for the policy.
  - The default value is an empty list (i.e. no aliases).
  - All alias names must follow the rules for the `name` option.
  - Aliases can be added to and removed from the list.
  - Aliases can be useful to retain support for old primary names if the primary name is changed.

- **`default = [true|false]` (optional)**
  - If `true` then this policy will be used when the client does not specify a policy.
  - The default value is `false`.
  - The default policy can be changed at any time, by setting `default = true` in the desired policy section.
  - If no policy is declared as the default and no other policies are defined, the policy with index `0` is set as the default;
  - Otherwise, exactly one policy must be declared default.
  - Deprecated policies cannot be declared the default.
  - See *Default versus Policy-0* for more information.

- **`deprecated = [true|false]` (optional)**
  - If `true` then new containers cannot be created using this policy.
  - The default value is `false`.
  - Any policy may be deprecated by adding the `deprecated` option to the desired policy section. However, a deprecated policy may not also be declared the default. Therefore, since there must always be a default policy, there must also always be at least one policy which is not deprecated.
  - See *Deprecating Policies* for more information.

- **`policy_type = [replication|erasure_coding]` (optional)**
  - The option `policy_type` is used to distinguish between different policy types.
  - The default value is `replication`.
  - When defining an EC policy use the value `erasure_coding`.

- **`diskfile_module = <entry point>` (optional)**

  - The option `diskfile_module` is used to load an alternate backend object storage plug-in architecture.
  - The default value is `egg:swift#replication.fs` or `egg:swift#erasure_coding.fs` depending on the policy type. The scheme and package name are optionals and default to `egg` and `swift`.

The EC policy type has additional required options. See *Using an Erasure Code Policy* for details.

The following is an example of a properly configured `swift.conf` file. See *Adding Storage Policies to an Existing SAIO* for full instructions on setting up an all-in-one with this example configuration.:

```
[swift-hash]
# random unique strings that can never change (DO NOT LOSE)
# Use only printable chars (python -c "import string; print(string.printable)
↪")
swift_hash_path_prefix = changeme
swift_hash_path_suffix = changeme

[storage-policy:0]
name = gold
aliases = yellow, orange
policy_type = replication
default = yes

[storage-policy:1]
name = silver
policy_type = replication
diskfile_module = replication.fs
deprecated = yes
```

Creating a ring

Once `swift.conf` is configured for a new policy, a new ring must be created. The ring tools are not policy name aware so its critical that the correct policy index be used when creating the new policys ring file. Additional object rings are created using `swift-ring-builder` in the same manner as the legacy ring except that `-N` is appended after the word `object` in the builder file name, where `N` matches the policy index used in `swift.conf`. So, to create the ring for policy index `1`:

```
swift-ring-builder object-1.builder create 10 3 1
```

Continue to use the same naming convention when using `swift-ring-builder` to add devices, rebalance etc. This naming convention is also used in the pattern for per-policy storage node data directories.

Note: The same drives can indeed be used for multiple policies and the details of how thats managed on disk will be covered in a later section, its important to understand the implications of such a configuration before setting one up. Make sure its really what you want to do, in many cases it will be, but in others maybe not.

Proxy server configuration (optional)

The *Proxy Server* configuration options related to read and write affinity may optionally be overridden for individual storage policies. See *Per policy configuration* for more details.

2.4.6 Using Policies

Using policies is very simple - a policy is only specified when a container is initially created. There are no other API changes. Creating a container can be done without any special policy information:

```
curl -v -X PUT -H 'X-Auth-Token: <your auth token>' \
    http://127.0.0.1:8080/v1/AUTH_test/myCont0
```

Which will result in a container created that is associated with the policy name gold assuming were using the swift.conf example from above. It would use gold because it was specified as the default. Now, when we put an object into this container, it will get placed on nodes that are part of the ring we created for policy gold.

If we wanted to explicitly state that we wanted policy gold the command would simply need to include a new header as shown below:

```
curl -v -X PUT -H 'X-Auth-Token: <your auth token>' \
    -H 'X-Storage-Policy: gold' http://127.0.0.1:8080/v1/AUTH_test/myCont0
```

And thats it! The application does not need to specify the policy name ever again. There are some illegal operations however:

- If an invalid (typo, non-existent) policy is specified: 400 Bad Request
- if you try to change the policy either via PUT or POST: 409 Conflict

If youd like to see how the storage in the cluster is being used, simply HEAD the account and youll see not only the cumulative numbers, as before, but per policy statistics as well. In the example below theres 3 objects total with two of them in policy gold and one in policy silver:

```
curl -i -X HEAD -H 'X-Auth-Token: <your auth token>' \
    http://127.0.0.1:8080/v1/AUTH_test
```

and your results will include (some output removed for readability):

```
X-Account-Container-Count: 3
X-Account-Object-Count: 3
X-Account-Bytes-Used: 21
X-Storage-Policy-Gold-Object-Count: 2
X-Storage-Policy-Gold-Bytes-Used: 14
X-Storage-Policy-Silver-Object-Count: 1
X-Storage-Policy-Silver-Bytes-Used: 7
```

2.4.7 Under the Hood

Now that weve explained a little about what Policies are and how to configure/use them, lets explore how Storage Policies fit in at the nuts-n-bolts level.

Parsing and Configuring

The module, *Storage Policy*, is responsible for parsing the `swift.conf` file, validating the input, and creating a global collection of configured policies via class *StoragePolicyCollection*. This collection is made up of policies of class *StoragePolicy*. The collection class includes handy functions for getting to a policy either by name or by index , getting info about the policies, etc. Theres also one very important function, *get_object_ring()*. Object rings are members of the *StoragePolicy* class and are actually not instantiated until the `load_ring()` method is called. Any caller anywhere in the code base that needs to access an object ring must use the `POLICIES` global singleton to access the *get_object_ring()* function and provide the policy index which will call `load_ring()` if needed; however, when starting request handling services such as the *Proxy Server* rings are proactively loaded to provide moderate protection against a mis-configuration resulting in a run time error. The global is instantiated when Swift starts and provides a mechanism to patch policies for the test code.

Middleware

Middleware can take advantage of policies through the `POLICIES` global and by importing *get_container_info()* to gain access to the policy index associated with the container in question. From the index it can then use the `POLICIES` singleton to grab the right ring. For example, *List Endpoints* is policy aware using the means just described. Another example is *Recon* which will report the md5 sums for all of the rings.

Proxy Server

The *Proxy Server* modules role in Storage Policies is essentially to make sure the correct ring is used as its member element. Before policies, the one object ring would be instantiated when the *Application* class was instantiated and could be overridden by test code via init parameter. With policies, however, there is no init parameter and the *Application* class instead depends on the `POLICIES` global singleton to retrieve the ring which is instantiated the first time its needed. So, instead of an object ring member of the *Application* class, there is an accessor function, *get_object_ring()*, that gets the ring from `POLICIES`.

In general, when any module running on the proxy requires an object ring, it does so via first getting the policy index from the cached container info. The exception is during container creation where it uses the policy name from the request header to look up policy index from the `POLICIES` global. Once the proxy has determined the policy index, it can use the *get_object_ring()* method described earlier to gain access to the correct ring. It then has the responsibility of passing the index information, not the policy name, on to the back-end servers via the header `X -Backend-Storage-Policy-Index`. Going the other way, the proxy also strips the index out of headers that go back to clients, and makes sure they only see the friendly policy names.

On Disk Storage

Policies each have their own directories on the back-end servers and are identified by their storage policy indexes. Organizing the back-end directory structures by policy index helps keep track of things and also allows for sharing of disks between policies which may or may not make sense depending on the needs of the provider. More on this later, but for now be aware of the following directory naming convention:

- `/objects` maps to objects associated with Policy-0
- `/objects-N` maps to storage policy index #N
- `/async_pending` maps to async pending update for Policy-0
- `/async_pending-N` maps to async pending update for storage policy index #N
- `/tmp` maps to the DiskFile temporary directory for Policy-0
- `/tmp-N` maps to the DiskFile temporary directory for policy index #N
- `/quarantined/objects` maps to the quarantine directory for Policy-0
- `/quarantined/objects-N` maps to the quarantine directory for policy index #N

Note that these directory names are actually owned by the specific Diskfile implementation, the names shown above are used by the default Diskfile.

Object Server

The *Object Server* is not involved with selecting the storage policy placement directly. However, because of how back-end directory structures are setup for policies, as described earlier, the object server modules do play a role. When the object server gets a `Diskfile`, it passes in the policy index and leaves the actual directory naming/structure mechanisms to `Diskfile`. By passing in the index, the instance of `Diskfile` being used will assure that data is properly located in the tree based on its policy.

For the same reason, the *Object Updater* also is policy aware. As previously described, different policies use different async pending directories so the updater needs to know how to scan them appropriately.

The *Object Replicator* is policy aware in that, depending on the policy, it may have to do drastically different things, or maybe not. For example, the difference in handling a replication job for 2x versus 3x is trivial; however, the difference in handling replication between 3x and erasure code is most definitely not. In fact, the term replication really isnt appropriate for some policies like erasure code; however, the majority of the framework for collecting and processing jobs is common. Thus, those functions in the replicator are leveraged for all policies and then there is policy specific code required for each policy, added when the policy is defined if needed.

The ssync functionality is policy aware for the same reason. Some of the other modules may not obviously be affected, but the back-end directory structure owned by `Diskfile` requires the policy index parameter. Therefore ssync being policy aware really means passing the policy index along. See *ssync_sender* and *ssync_receiver* for more information on ssync.

For `Diskfile` itself, being policy aware is all about managing the back-end structure using the provided policy index. In other words, callers who get a `Diskfile` instance provide a policy index and `Diskfile`s job is to keep data separated via this index (however it chooses) such that policies can share the same media/nodes if desired. The included implementation of `Diskfile` lays out the directory structure described earlier but thats owned within `Diskfile`; external modules have no visibility into that detail. A common function is provided to map various directory names and/or strings based on their policy index.

For example `Diskfile` defines *`get_data_dir()`* which builds off of a generic *`get_policy_string()`* to consistently build policy aware strings for various usage.

Container Server

The *Container Server* plays a very important role in Storage Policies, it is responsible for handling the assignment of a policy to a container and the prevention of bad things like changing policies or picking the wrong policy to use when nothing is specified (recall earlier discussion on Policy-0 versus default).

The *Container Updater* is policy aware, however its job is very simple, to pass the policy index along to the *Account Server* via a request header.

The *Container Backend* is responsible for both altering existing DB schema as well as assuring new DBs are created with a schema that supports storage policies. The on-demand migration of container schemas allows Swift to upgrade without downtime (sqlites alter statements are fast regardless of row count). To support rolling upgrades (and downgrades) the incompatible schema changes to the `container_stat` table are made to a `container_info` table, and the `container_stat` table is replaced with a view that includes an `INSTEAD OF UPDATE` trigger which makes it behave like the old table.

The policy index is stored here for use in reporting information about the container as well as managing split-brain scenario induced discrepancies between containers and their storage policies. Furthermore, during split-brain, containers must be prepared to track object updates from multiple policies so the object table also includes a `storage_policy_index` column. Per-policy object counts and bytes are updated in the `policy_stat` table using `INSERT` and `DELETE` triggers similar to the pre-policy triggers that updated `container_stat` directly.

The *Container Replicator* daemon will pro-actively migrate legacy schemas as part of its normal consistency checking process when it updates the `reconciler_sync_point` entry in the `container_info` table. This ensures that read heavy containers which do not encounter any writes will still get migrated to be fully compatible with the post-storage-policy queries without having to fall back and retry queries with the legacy schema to service container read requests.

The *Container Sync* functionality only needs to be policy aware in that it accesses the object rings. Therefore, it needs to pull the policy index out of the container information and use it to select the appropriate object ring from the `POLICIES` global.

Account Server

The *Account Server*s role in Storage Policies is really limited to reporting. When a HEAD request is made on an account (see example provided earlier), the account server is provided with the storage policy index and builds the `object_count` and `byte_count` information for the client on a per policy basis.

The account servers are able to report per-storage-policy object and byte counts because of some policy specific DB schema changes. A policy specific table, `policy_stat`, maintains information on a per policy basis (one row per policy) in the same manner in which the `account_stat` table does. The `account_stat` table still serves the same purpose and is not replaced by `policy_stat`, it holds the total account stats whereas `policy_stat` just has the break downs. The backend is also responsible for migrating pre-storage-policy accounts by altering the DB schema and populating the `policy_stat` table for Policy-0 with current `account_stat` data at that point in time.

The per-storage-policy object and byte counts are not updated with each object PUT and DELETE request, instead container updates to the account server are performed asynchronously by the `swift-container-updater`.

Upgrading and Confirming Functionality

Upgrading to a version of Swift that has Storage Policy support is not difficult, in fact, the cluster administrator isnt required to make any special configuration changes to get going. Swift will automatically begin using the existing object ring as both the default ring and the Policy-0 ring. Adding the declaration of policy 0 is totally optional and in its absence, the name given to the implicit policy 0 will be Policy-0. Lets say for testing purposes that you wanted to take an existing cluster that already has lots of data on it and upgrade to Swift with Storage Policies. From there you want to go ahead and create a policy and test a few things out. All you need to do is:

1. Upgrade all of your Swift nodes to a policy-aware version of Swift
2. Define your policies in `/etc/swift/swift.conf`
3. Create the corresponding object rings
4. Create containers and objects and confirm their placement is as expected

For a specific example that takes you through these steps, please see *Adding Storage Policies to an Existing SAIO*

Note: If you downgrade from a Storage Policy enabled version of Swift to an older version that doesnt support policies, you will not be able to access any data stored in policies other than the policy with index 0 but those objects WILL appear in container listings (possibly as duplicates if there was a network partition and un-reconciled objects). It is EXTREMELY important that you perform any necessary integration testing on the upgraded deployment before enabling an additional storage policy to ensure a consistent API experience for your clients. DO NOT downgrade to a version of Swift that does not support storage policies once you expose multiple storage policies.

2.5 The Account Reaper

The Account Reaper removes data from deleted accounts in the background.

An account is marked for deletion by a reseller issuing a DELETE request on the accounts storage URL. This simply puts the value DELETED into the status column of the account_stat table in the account database (and replicas), indicating the data for the account should be deleted later.

There is normally no set retention time and no undelete; it is assumed the reseller will implement such features and only call DELETE on the account once it is truly desired the accounts data be removed. However, in order to protect the Swift cluster accounts from an improper or mistaken delete request, you can set a delay_reaping value in the [account-reaper] section of the account-server.conf to delay the actual deletion of data. At this time, there is no utility to undelete an account; one would have to update the account database replicas directly, setting the status column to an empty string and updating the put_timestamp to be greater than the delete_timestamp. (On the TODO list is writing a utility to perform this task, preferably through a REST call.)

The account reaper runs on each account server and scans the server occasionally for account databases marked for deletion. It will only trigger on accounts that server is the primary node for, so that multiple account servers arent all trying to do the same work at the same time. Using multiple servers to delete one account might improve deletion speed, but requires coordination so they arent duplicating effort. Speed really isnt as much of a concern with data deletion and large accounts arent deleted that often.

The deletion process for an account itself is pretty straightforward. For each container in the account, each object is deleted and then the container is deleted. Any deletion requests that fail wont stop the overall process, but will cause the overall process to fail eventually (for example, if an object delete times out, the container wont be able to be deleted later and therefore the account wont be deleted either). The overall process continues even on a failure so that it doesnt get hung up reclaiming cluster space because of one troublesome spot. The account reaper will keep trying to delete an account until it eventually becomes empty, at which point the database reclaim process within the db_replicator will eventually remove the database files.

Sometimes a persistent error state can prevent some object or container from being deleted. If this happens, you will see a message such as Account <name> has not been reaped since <date> in the log. You can control when this is logged with the reap_warn_after value in the [account-reaper] section of the account-server.conf file. By default this is 30 days.

2.5.1 History

At first, a simple approach of deleting an account through completely external calls was considered as it required no changes to the system. All data would simply be deleted in the same way the actual user would, through the public REST API. However, the downside was that it would use proxy resources and log everything when it didnt really need to. Also, it would likely need a dedicated server or two, just for issuing the delete requests.

A completely bottom-up approach was also considered, where the object and container servers would occasionally scan the data they held and check if the account was deleted, removing the data if so. The upside was the speed of reclamation with no impact on the proxies or logging, but the downside was that nearly 100% of the scanning would result in no action creating a lot of I/O load for no reason.

A more container server centric approach was also considered, where the account server would mark all the containers for deletion and the container servers would delete the objects in each container and then themselves. This has the benefit of still speedy reclamation for accounts with a lot of containers, but has the downside of a pretty big load spike. The process could be slowed down to alleviate the load spike possibility, but then the benefit of speedy reclamation is lost and whats left is just a more complex process. Also, scanning all the containers for those marked for deletion when the majority wouldnt be seemed wasteful. The db_replicator could do this work while performing its replication scan, but it would have to spawn and track deletion processes which seemed needlessly complex.

In the end, an account server centric approach seemed best, as described above.

2.6 The Auth System

2.6.1 Overview

Swift supports a number of auth systems that share the following common characteristics:

- The authentication/authorization part can be an external system or a subsystem run within Swift as WSGI middleware
- The user of Swift passes in an auth token with each request
- Swift validates each token with the external auth system or auth subsystem and caches the result
- The token does not change from request to request, but does expire

The token can be passed into Swift using the X-Auth-Token or the X-Storage-Token header. Both have the same format: just a simple string representing the token. Some auth systems use UUID tokens, some an MD5 hash of something unique, some use something else but the salient point is that the token is a string which can be sent as-is back to the auth system for validation.

Swift will make calls to the auth system, giving the auth token to be validated. For a valid token, the auth system responds with an overall expiration time in seconds from now. To avoid the overhead in validating the same token over and over again, Swift will cache the token for a configurable time, but no longer than the expiration time.

The Swift project includes two auth systems:

- *TempAuth*
- *Keystone Auth*

It is also possible to write your own auth system as described in *Extending Auth*.

2.6.2 TempAuth

TempAuth is used primarily in Swifts functional test environment and can be used in other test environments (such as *SAIO (Swift All In One)*). It is not recommended to use TempAuth in a production system. However, TempAuth is fully functional and can be used as a model to develop your own auth system.

TempAuth has the concept of admin and non-admin users within an account. Admin users can do anything within the account. Non-admin users can only perform read operations. However, some privileged metadata such as X-Container-Sync-Key is not accessible to non-admin users.

Users with the special group `.reseller_admin` can operate on any account. For an example usage please see `swift.common.middleware.tempauth`. If a request is coming from a reseller the auth system sets the request environ reseller_request to True. This can be used by other middlewares.

Other users may be granted the ability to perform operations on an account or container via ACLs. TempAuth supports two types of ACL:

- Per container ACLs based on the containers `X-Container-Read` and `X-Container-Write` metadata. See *Container ACLs* for more information.
- Per account ACLs based on the accounts `X-Account-Access-Control` metadata. For more information see *Account ACLs*.

TempAuth will now allow OPTIONS requests to go through without a token.

The TempAuth middleware is responsible for creating its own tokens. A user makes a request containing their username and password and TempAuth responds with a token. This token is then used to perform subsequent requests on the users account, containers and objects.

2.6.3 Keystone Auth

Swift is able to authenticate against OpenStack Keystone. In this environment, Keystone is responsible for creating and validating tokens. The *KeystoneAuth* middleware is responsible for implementing the auth system within Swift as described here.

The *KeystoneAuth* middleware supports per container based ACLs on the containers `X-Container-Read` and `X-Container-Write` metadata. For more information see *Container ACLs*.

The account-level ACL is not supported by Keystone auth.

In order to use the `keystoneauth` middleware the `auth_token` middleware from KeystoneMiddleware will need to be configured.

The `authtoken` middleware performs the authentication token validation and retrieves actual user authentication information. It can be found in the KeystoneMiddleware distribution.

The *KeystoneAuth* middleware performs authorization and mapping the Keystone roles to Swifts ACLs.

Configuring Swift to use Keystone

Configuring Swift to use Keystone is relatively straightforward. The first step is to ensure that you have the `auth_token` middleware installed. It can either be dropped in your python path or installed via the KeystoneMiddleware package.

You need at first make sure you have a service endpoint of type `object-store` in Keystone pointing to your Swift proxy. For example having this in your `/etc/keystone/default_catalog.templates`

```
catalog.RegionOne.object_store.name = Swift Service
catalog.RegionOne.object_store.publicURL = http://swiftproxy:8080/v1/AUTH_
↪$(tenant_id)s
catalog.RegionOne.object_store.adminURL = http://swiftproxy:8080/
catalog.RegionOne.object_store.internalURL = http://swiftproxy:8080/v1/AUTH_
↪$(tenant_id)s
```

On your Swift proxy server you will want to adjust your main pipeline and add auth_token and keystoneauth in your `/etc/swift/proxy-server.conf` like this

```
[pipeline:main]
pipeline = [....] authtoken keystoneauth proxy-logging proxy-server
```

add the configuration for the authtoken middleware:

```
[filter:authtoken]
paste.filter_factory = keystonemiddleware.auth_token:filter_factory
www_authenticate_uri = http://keystonehost:5000/
auth_url = http://keystonehost:5000/
auth_plugin = password
project_domain_id = default
user_domain_id = default
project_name = service
username = swift
password = password
```

(continues on next page)

(continued from previous page)

```
cache = swift.cache
include_service_catalog = False
delay_auth_decision = True
```

The actual values for these variables will need to be set depending on your situation, but in short:

- `www_authenticate_uri` should point to a Keystone service from which users may retrieve tokens. This value is used in the *WWW-Authenticate* header that auth_token sends with any denial response.
- `auth_url` points to the Keystone Admin service. This information is used by the middleware to actually query Keystone about the validity of the authentication tokens. It is not necessary to append any Keystone API version number to this URI.
- The auth credentials (`project_domain_id`, `user_domain_id`, `username`, `project_name`, `password`) will be used to retrieve an admin token. That token will be used to authorize user tokens behind the scenes. These credentials must match the Keystone credentials for the Swift service. The example values shown here assume a user named swift with admin role on a project named service, both being in the Keystone domain with id default. Refer to the KeystoneMiddleware documentation for other examples.
- `cache` is set to `swift.cache`. This means that the middleware will get the Swift memcache from the request environment.
- `include_service_catalog` defaults to `True` if not set. This means that when validating a token, the service catalog is retrieved and stored in the `X-Service-Catalog` header. Since Swift does not use the `X-Service-Catalog` header, there is no point in getting the service catalog. We recommend you set `include_service_catalog` to `False`.

Note: The authtoken config variable `delay_auth_decision` must be set to `True`. The default is `False`, but that breaks public access, *StaticWeb*, *FormPost*, *TempURL*, and authenticated capabilities requests (using *Discoverability*).

---

and you can finally add the keystoneauth configuration. Here is a simple configuration:

```
[filter:keystoneauth]
use = egg:swift#keystoneauth
operator_roles = admin, swiftoperator
```

Use an appropriate list of roles in operator_roles. For example, in some systems, the role `_member_` or `Member` is used to indicate that the user is allowed to operate on project resources.

OpenStack Service Using Composite Tokens

Some OpenStack services such as Cinder and Glance may use a service account. In this mode, you configure a separate account where the service stores project data that it manages. This account is not used directly by the end-user. Instead, all access is done through the service.

To access the service account, the service must present two tokens: one from the end-user and another from its own service user. Only when both tokens are present can the account be accessed. This section describes how to set the configuration options to correctly control access to both the normal and service accounts.

In this example, end users use the `AUTH_` prefix in account names, whereas services use the `SERVICE_` prefix:

```
[filter:keystoneauth]
use = egg:swift#keystoneauth
reseller_prefix = AUTH, SERVICE
operator_roles = admin, swiftoperator
SERVICE_service_roles = service
```

The actual values for these variable will need to be set depending on your situation as follows:

- The first item in the reseller_prefix list must match Keystones endpoint (see `/etc/keystone/default_catalog.templates` above). Normally this is `AUTH`.
- The second item in the reseller_prefix list is the prefix used by the OpenStack services(s). You must configure this value (`SERVICE` in the example) with whatever the other OpenStack service(s) use.
- Set the operator_roles option to contain a role or roles that end-users have on projects they use.
- Set the SERVICE_service_roles value to a role or roles that only the OpenStack service user has. Do not use a role that is assigned to normal end users. In this example, the role `service` is used. The service user is granted this role to a *single* project only. You do not need to make the service user a member of every project.

This configuration works as follows:

- The end-user presents a user token to an OpenStack service. The service then makes a Swift request to the account with the `SERVICE` prefix.
- The service forwards the original user token with the request. It also adds its own service token.
- Swift validates both tokens. When validated, the user token gives the `admin` or `swiftoperator` role(s). When validated, the service token gives the `service` role.
- Swift interprets the above configuration as follows:
  - Did the user token provide one of the roles listed in operator_roles?
  - Did the service token have the `service` role as described by the `SERVICE_service_roles` options.
- If both conditions are met, the request is granted. Otherwise, Swift rejects the request.

In the above example, all services share the same account. You can separate each service into its own account. For example, the following provides a dedicated account for each of the Glance and Cinder services. In addition, you must assign the `glance_service` and `cinder_service` to the appropriate service users:

```
[filter:keystoneauth]
use = egg:swift#keystoneauth
reseller_prefix = AUTH, IMAGE, VOLUME
operator_roles = admin, swiftoperator
IMAGE_service_roles = glance_service
VOLUME_service_roles = cinder_service
```

Access control using keystoneauth

By default the only users able to perform operations (e.g. create a container) on an account are those having a Keystone role for the corresponding Keystone project that matches one of the roles specified in the `operator_roles` option.

Users who have one of the `operator_roles` will be able to set container ACLs to grant other users permission to read and/or write objects in specific containers, using `X-Container-Read` and `X-Container-Write` headers respectively. In addition to the ACL formats described *here*, keystoneauth supports ACLs using the format:

```
other_project_id:other_user_id.
```

where `other_project_id` is the UUID of a Keystone project and `other_user_id` is the UUID of a Keystone user. This will allow the other user to access a container provided their token is scoped on the other project. Both `other_project_id` and `other_user_id` may be replaced with the wildcard character * which will match any project or user respectively.

Be sure to use Keystone UUIDs rather than names in container ACLs.

Note: For backwards compatibility, keystoneauth will by default grant container ACLs expressed as `other_project_name:other_user_name` (i.e. using Keystone names rather than UUIDs) in the special case when both the other project and the other user are in Keystones default domain and the project being accessed is also in the default domain.

For further information see *KeystoneAuth*

Users with the Keystone role defined in `reseller_admin_role` (`ResellerAdmin` by default) can operate on any account. The auth system sets the request environ reseller_request to True if a request is coming from a user with this role. This can be used by other middlewares.

Troubleshooting tips for keystoneauth deployment

Some common mistakes can result in API requests failing when first deploying keystone with Swift:

- Incorrect configuration of the Swift endpoint in the Keystone service.

  By default, keystoneauth expects the account part of a URL to have the form `AUTH_<keystone_project_id>`. Sometimes the `AUTH_` prefix is missed when configuring Swift endpoints in Keystone, as described in the Install Guide. This is easily diagnosed by inspecting the proxy-server log file for a failed request URL and checking that the URL includes the `AUTH_` prefix (or whatever reseller prefix may have been configured for keystoneauth):

```
GOOD:
proxy-server: 127.0.0.1 127.0.0.1 07/Sep/2016/16/06/58 HEAD /v1/AUTH_
↪cfb8d9d45212408b90bc0776117aec9e HTTP/1.0 204 ...

BAD:
proxy-server: 127.0.0.1 127.0.0.1 07/Sep/2016/16/07/35 HEAD /v1/
↪cfb8d9d45212408b90bc0776117aec9e HTTP/1.0 403 ...
```

- Incorrect configuration of the `authtoken` middleware options in the Swift proxy server.

  The `authtoken` middleware communicates with the Keystone service to validate tokens that are presented with client requests. To do this `authtoken` must authenticate itself with Keystone using the credentials configured in the `[filter:authtoken]` section of `/etc/swift/proxy-server.conf`. Errors in these credentials can result in `authtoken` failing to validate tokens and may be revealed in the proxy server logs by a message such as:

```
proxy-server: Identity server rejected authorization
```

  Note: More detailed log messaging may be seen by setting the `authtoken` option `log_level = debug`.

  The `authtoken` configuration options may be checked by attempting to use them to communicate directly with Keystone using an `openstack` command line. For example, given the `authtoken` configuration sample shown in *Configuring Swift to use Keystone*, the following command should return a service catalog:

```
openstack --os-identity-api-version=3 --os-auth-url=http://
↪keystonehost:5000/ \
    --os-username=swift --os-user-domain-id=default \
    --os-project-name=service --os-project-domain-id=default \
    --os-password=password catalog show object-store
```

  If this `openstack` command fails then it is likely that there is a problem with the `authtoken` configuration.

2.6.4 Extending Auth

TempAuth is written as wsgi middleware, so implementing your own auth is as easy as writing new wsgi middleware, and plugging it in to the proxy server.

See *Auth Server and Middleware* for detailed information on extending the auth system.

2.7 Access Control Lists (ACLs)

Normally to create, read and modify containers and objects, you must have the appropriate roles on the project associated with the account, i.e., you must be the owner of the account. However, an owner can grant access to other users by using an Access Control List (ACL).

There are two types of ACLs:

- *Container ACLs*. These are specified on a container and apply to that container only and the objects in the container.
- *Account ACLs*. These are specified at the account level and apply to all containers and objects in the account.

2.7.1 Container ACLs

Container ACLs are stored in the `X-Container-Write` and `X-Container-Read` metadata. The scope of the ACL is limited to the container where the metadata is set and the objects in the container. In addition:

- `X-Container-Write` grants the ability to perform PUT, POST and DELETE operations on objects within a container. It does not grant the ability to perform POST or DELETE operations on the container itself. Some ACL elements also grant the ability to perform HEAD or GET operations on the container.
- `X-Container-Read` grants the ability to perform GET and HEAD operations on objects within a container. Some of the ACL elements also grant the ability to perform HEAD or GET operations on the container itself. However, a container ACL does not allow access to privileged metadata (such as `X-Container-Sync-Key`).

Container ACLs use the V1 ACL syntax which is a comma separated string of elements as shown in the following example:

```
.r:*,.rlistings,7ec59e87c6584c348b563254aae4c221:*
```

Spaces may occur between elements as shown in the following example:

```
.r : *, .rlistings, 7ec59e87c6584c348b563254aae4c221:*
```

However, these spaces are removed from the value stored in the `X-Container-Write` and `X-Container-Read` metadata. In addition, the `.r:` string can be written as `.referrer:`, but is stored as `.r:`.

While all auth systems use the same syntax, the meaning of some elements is different because of the different concepts used by different auth systems as explained in the following sections:

- *Common ACL Elements*

- *Keystone Auth ACL Elements*
- *TempAuth ACL Elements*

Common ACL Elements

The following table describes elements of an ACL that are supported by both Keystone auth and TempAuth. These elements should only be used with `X-Container-Read` (with the exception of `.rlistings`, an error will occur if used with `X-Container-Write`):

Element	Description
.r:*	Any user has access to objects. No token is required in the request.
.r:<referrer>	The referrer is granted access to objects. The referrer is identified by the `Referer` request header in the request. No token is required.
.r:-<referrer>	This syntax (with - prepended to the referrer) is supported. However, it does not deny access if another element (e.g., `.r:*`) grants access.
.rlistings	Any user can perform a HEAD or GET operation on the container provided the user also has read access on objects (e.g., also has `.r:*` or `.r:<referrer>`. No token is required.

Keystone Auth ACL Elements

The following table describes elements of an ACL that are supported only by Keystone auth. Keystone auth also supports the elements described in *Common ACL Elements*.

A token must be included in the request for any of these ACL elements to take effect.

Element	Description
<project-id>:<user-id>	The specified user, provided a token scoped to the project is included in the request, is granted access. Access to the container is also granted when used in `X-Container-Read`.
<project-id>:*	Any user with a role in the specified Keystone project has access. A token scoped to the project must be included in the request. Access to the container is also granted when used in `X-Container-Read`.
*:<user-id>	The specified user has access. A token for the user (scoped to any project) must be included in the request. Access to the container is also granted when used in `X-Container-Read`.
:	Any user has access. Access to the container is also granted when used in `X-Container-Read`. The `*:*` element differs from the `.r:*` element because `*:*` requires that a valid token is included in the request whereas `.r:*` does not require a token. In addition, `.r:*` does not grant access to the container listing.
<role_name>	A user with the specified role *name* on the project within which the container is stored is granted access. A user token scoped to the project must be included in the request. Access to the container is also granted when used in `X-Container-Read`.

Note: Keystone project (tenant) or user *names* (i.e., `<project-name>:<user-name>`) must no longer be used because with the introduction of domains in Keystone, names are not globally unique. You should use user and project *ids* instead.

For backwards compatibility, ACLs using names will be granted by keystoneauth when it can be established that the grantee project, the grantee user and the project being accessed are either not yet in a domain (e.g. the `X-Auth-Token` has been obtained via the Keystone V2 API) or are all in the default domain to which legacy accounts would have been migrated.

TempAuth ACL Elements

The following table describes elements of an ACL that are supported only by TempAuth. TempAuth auth also supports the elements described in *Common ACL Elements*.

Element	Description
<username>	The named user is granted access. The wildcard (*) character is not supported. A token from the user must be included in the request.

2.7.2 Container ACL Examples

Container ACLs may be set by including `X-Container-Write` and/or `X-Container-Read` headers with a PUT or a POST request to the container URL. The following examples use the `swift` command line client which support these headers being set via its `--write-acl` and `--read-acl` options.

Example: Public Container

The following allows anybody to list objects in the `www` container and download objects. The users do not need to include a token in their request. This ACL is commonly referred to as making the container public. It is useful when used with *StaticWeb*:

```
swift post www --read-acl ".r:*,.rlistings"
```

Example: Shared Writable Container

The following allows anybody to upload or download objects. However, to download an object, the exact name of the object must be known since users cannot list the objects in the container. The users must include a Keystone token in the upload request. However, it does not need to be scoped to the project associated with the container:

```
swift post www --read-acl ".r:*" --write-acl "*:*"
```

Example: Sharing a Container with Project Members

The following allows any member of the `77b8f82565f14814bece56e50c4c240f` project to upload and download objects or to list the contents of the `www` container. A token scoped to the `77b8f82565f14814bece56e50c4c240f` project must be included in the request:

```
swift post www --read-acl "77b8f82565f14814bece56e50c4c240f:*" \
               --write-acl "77b8f82565f14814bece56e50c4c240f:*"
```

Example: Sharing a Container with Users having a specified Role

The following allows any user that has been assigned the `my_read_access_role` on the project within which the `www` container is stored to download objects or to list the contents of the `www` container. A user token scoped to the project must be included in the download or list request:

```
swift post www --read-acl "my_read_access_role"
```

Example: Allowing a Referrer Domain to Download Objects

The following allows any request from the `example.com` domain to access an object in the container:

```
swift post www --read-acl ".r:.example.com"
```

However, the request from the user **must** contain the appropriate *Referer* header as shown in this example request:

```
curl -i $publicURL/www/document --head -H "Referer: http://www.example.com/
↪index.html"
```

Note: The *Referer* header is included in requests by many browsers. However, since it is easy to create a request with any desired value in the *Referer* header, the referrer ACL has very weak security.

Example: Sharing a Container with Another User

Sharing a Container with another user requires the knowledge of few parameters regarding the users.

The sharing user must know:

- the `OpenStack user id` of the other user

The sharing user must communicate to the other user:

- the name of the shared container
- the `OS_STORAGE_URL`

Usually the `OS_STORAGE_URL` is not exposed directly to the user because the `swift client` by default automatically construct the `OS_STORAGE_URL` based on the User credential.

We assume that in the current directory there are the two client environment script for the two users `sharing.openrc` and `other.openrc`.

The `sharing.openrc` should be similar to the following:

```
export OS_USERNAME=sharing
# WARNING: Save the password in clear text only for testing purposes
export OS_PASSWORD=password
export OS_TENANT_NAME=projectName
export OS_AUTH_URL=https://identityHost:portNumber/v2.0
# The following lines can be omitted
export OS_TENANT_ID=tenantIDString
export OS_REGION_NAME=regionName
export OS_CACERT=/path/to/cacertFile
```

The `other.openrc` should be similar to the following:

```
export OS_USERNAME=other
# WARNING: Save the password in clear text only for testing purposes
export OS_PASSWORD=otherPassword
export OS_TENANT_NAME=otherProjectName
export OS_AUTH_URL=https://identityHost:portNumber/v2.0
# The following lines can be omitted
export OS_TENANT_ID=tenantIDString
export OS_REGION_NAME=regionName
export OS_CACERT=/path/to/cacertFile
```

For more information see using the OpenStack RC file

First we figure out the other user id:

```
. other.openrc
OUID="$(openstack user show --format json "${OS_USERNAME}" | jq -r .id)"
```

or alternatively:

```
. other.openrc
OUID="$(openstack token issue -f json | jq -r .user_id)"
```

Then we figure out the storage url of the sharing user:

```
sharing.openrc
SURL="$(swift auth | awk -F = '/OS_STORAGE_URL/ {print $2}')"
```

Running as the sharing user create a shared container named `shared` in read-only mode with the other user using the proper acl:

```
sharing.openrc
swift post --read-acl "*:${OUID}" shared
```

Running as the sharing user create and upload a test file:

```
touch void
swift upload shared void
```

Running as the other user list the files in the `shared` container:

```
other.openrc
swift --os-storage-url="${SURL}" list shared
```

Running as the other user download the `shared` container in the `/tmp` directory:

```
cd /tmp
swift --os-storage-url="${SURL}" download shared
```

2.7.3 Account ACLs

Note: Account ACLs are not currently supported by Keystone auth

The `X-Account-Access-Control` header is used to specify account-level ACLs in a format specific to the auth system. These headers are visible and settable only by account owners (those for whom `swift_owner` is true). Behavior of account ACLs is auth-system-dependent. In the case of TempAuth, if an authenticated user has membership in a group which is listed in the ACL, then the user is allowed the access level of that ACL.

Account ACLs use the V2 ACL syntax, which is a JSON dictionary with keys named admin, read-write, and read-only. (Note the case sensitivity.) An example value for the `X-Account-Access-Control` header looks like this, where `a`, `b` and `c` are user names:

```
{"admin":["a","b"],"read-only":["c"]}
```

Keys may be absent (as shown in above example).

The recommended way to generate ACL strings is as follows:

```
from swift.common.middleware.acl import format_acl
acl_data = { 'admin': ['alice'], 'read-write': ['bob', 'carol'] }
acl_string = format_acl(version=2, acl_dict=acl_data)
```

Using the `format_acl()` method will ensure that JSON is encoded as ASCII (using e.g. u1234 for Unicode). While its permissible to manually send `curl` commands containing `X-Account-Access-Control` headers, you should exercise caution when doing so, due to the potential for human error.

Within the JSON dictionary stored in `X-Account-Access-Control`, the keys have the following meanings:

Access Level	Description
read-only	These identities can read *everything* (except privileged headers) in the account. Specifically, a user with read-only account access can get a list of containers in the account, list the contents of any container, retrieve any object, and see the (non-privileged) headers of the account, any container, or any object.
read-write	These identities can read or write (or create) any container. A user with read-write account access can create new containers, set any unprivileged container headers, overwrite objects, delete containers, etc. A read-write user can NOT set account headers (or perform any PUT/POST/DELETE requests on the account).
admin	These identities have swift_owner privileges. A user with admin account access can do anything the account owner can, including setting account headers and any privileged headers and thus granting read-only, read-write, or admin access to other users.

For more details, see *swift.common.middleware.tempauth*. For details on the ACL format, see *swift.common.middleware.acl*.

2.8 Replication

Because each replica in Swift functions independently, and clients generally require only a simple majority of nodes responding to consider an operation successful, transient failures like network partitions can quickly cause replicas to diverge. These differences are eventually reconciled by asynchronous, peer-to-peer replicator processes. The replicator processes traverse their local filesystems, concurrently performing operations in a manner that balances load across physical disks.

Replication uses a push model, with records and files generally only being copied from local to remote replicas. This is important because data on the node may not belong there (as in the case of handoffs and ring changes), and a replicator cant know what data exists elsewhere in the cluster that it should pull in. Its the duty of any node that contains data to ensure that data gets to where it belongs. Replica placement is handled by the ring.

Every deleted record or file in the system is marked by a tombstone, so that deletions can be replicated alongside creations. The replication process cleans up tombstones after a time period known as the consistency window. The consistency window encompasses replication duration and how long transient failure can remove a node from the cluster. Tombstone cleanup must be tied to replication to reach replica convergence.

If a replicator detects that a remote drive has failed, the replicator uses the get_more_nodes interface for the ring to choose an alternate node with which to synchronize. The replicator can maintain desired levels of replication in the face of disk failures, though some replicas may not be in an immediately usable location. Note that the replicator doesnt maintain desired levels of replication when other failures, such as entire node failures, occur because most failure are transient.

Replication is an area of active development, and likely rife with potential improvements to speed and correctness.

There are two major classes of replicator - the db replicator, which replicates accounts and containers, and the object replicator, which replicates object data.

2.8.1 DB Replication

The first step performed by db replication is a low-cost hash comparison to determine whether two replicas already match. Under normal operation, this check is able to verify that most databases in the system are already synchronized very quickly. If the hashes differ, the replicator brings the databases in sync by sharing records added since the last sync point.

This sync point is a high water mark noting the last record at which two databases were known to be in sync, and is stored in each database as a tuple of the remote database id and record id. Database ids are unique amongst all replicas of the database, and record ids are monotonically increasing integers. After all new records have been pushed to the remote database, the entire sync table of the local database is pushed, so the remote database can guarantee that it is in sync with everything with which the local database has previously synchronized.

If a replica is found to be missing entirely, the whole local database file is transmitted to the peer using rsync(1) and vested with a new unique id.

In practice, DB replication can process hundreds of databases per concurrency setting per second (up to the number of available CPUs or disks) and is bound by the number of DB transactions that must be performed.

2.8.2 Object Replication

The initial implementation of object replication simply performed an rsync to push data from a local partition to all remote servers it was expected to exist on. While this performed adequately at small scale, replication times skyrocketed once directory structures could no longer be held in RAM. We now use a modification of this scheme in which a hash of the contents for each suffix directory is saved to a per-partition hashes file. The hash for a suffix directory is invalidated when the contents of that suffix directory are modified.

The object replication process reads in these hash files, calculating any invalidated hashes. It then transmits the hashes to each remote server that should hold the partition, and only suffix directories with differing hashes on the remote server are rsynced. After pushing files to the remote server, the replication process notifies it to recalculate hashes for the rsynced suffix directories.

Performance of object replication is generally bound by the number of uncached directories it has to traverse, usually as a result of invalidated suffix directory hashes. Using write volume and partition counts from our running systems, it was designed so that around 2% of the hash space on a normal node will be invalidated per day, which has experimentally given us acceptable replication speeds.

Work continues with a new ssync method where rsync is not used at all and instead all-Swift code is used to transfer the objects. At first, this ssync will just strive to emulate the rsync behavior. Once deemed stable it will open the way for future improvements in replication since well be able to easily add code in the replication path instead of trying to alter the rsync code base and distributing such modifications.

One of the first improvements planned is an index.db that will replace the hashes.pkl. This will allow quicker updates to that data as well as more streamlined queries. Quite likely well implement a better scheme than the current one hashes.pkl uses (hash-trees, that sort of thing).

Another improvement planned all along the way is separating the local disk structure from the protocol path structure. This separation will allow ring resizing at some point, or at least ring-doubling.

Note that for objects being stored with an Erasure Code policy, the replicator daemon is not involved. Instead, the reconstructor is used by Erasure Code policies and is analogous to the replicator for Repli-

cation type policies. See *Erasure Code Support* for complete information on both Erasure Code support as well as the reconstructor.

2.8.3 Hashes.pkl

The hashes.pkl file is a key element for both replication and reconstruction (for Erasure Coding). Both daemons use this file to determine if any kind of action is required between nodes that are participating in the durability scheme. The file itself is a pickled dictionary with slightly different formats depending on whether the policy is Replication or Erasure Code. In either case, however, the same basic information is provided between the nodes. The dictionary contains a dictionary where the key is a suffix directory name and the value is the MD5 hash of the directory listing for that suffix. In this manner, the daemon can quickly identify differences between local and remote suffix directories on a per partition basis as the scope of any one hashes.pkl file is a partition directory.

For Erasure Code policies, there is a little more information required. An objects hash directory may contain multiple fragments of a single object in the event that the node is acting as a handoff or perhaps if a rebalance is underway. Each fragment of an object is stored with a fragment index, so the hashes.pkl for an Erasure Code partition will still be a dictionary keyed on the suffix directory name, however, the value is another dictionary keyed on the fragment index with subsequent MD5 hashes for each one as values. Some files within an object hash directory dont require a fragment index so None is used to represent those. Below are examples of what these dictionaries might look like.

Replication hashes.pkl:

```
{'a43': '72018c5fbfae934e1f56069ad4425627',
 'b23': '12348c5fbfae934e1f56069ad4421234'}
```

Erasure Code hashes.pkl:

```
{'a43': {None: '72018c5fbfae934e1f56069ad4425627',
         2: 'b6dd6db937cb8748f50a5b6e4bc3b808'},
 'b23': {None: '12348c5fbfae934e1f56069ad4421234',
         1: '45676db937cb8748f50a5b6e4bc34567'}}
```

2.8.4 Dedicated replication network

Swift has support for using dedicated network for replication traffic. For more information see *Overview of dedicated replication network*.

2.9 Rate Limiting

Rate limiting in Swift is implemented as a pluggable middleware. Rate limiting is performed on requests that result in database writes to the account and container sqlite dbs. It uses memcached and is dependent on the proxy servers having highly synchronized time. The rate limits are limited by the accuracy of the proxy server clocks.

2.9.1 Configuration

All configuration is optional. If no account or container limits are provided there will be no rate limiting. Configuration available:

Option	Default	Description
clock_accuracy	1000	Represents how accurate the proxy servers system clocks are with each other. 1000 means that all the proxies clock are accurate to each other within 1 millisecond. No ratelimit should be higher than the clock accuracy.
max_sleep_time_seconds	60	App will immediately return a 498 response if the necessary sleep time ever exceeds the given max_sleep_time_seconds.
log_sleep_time_seconds	0	To allow visibility into rate limiting set this value > 0 and all sleeps greater than the number will be logged.
rate_buffer_seconds	5	Number of seconds the rate counter can drop and be allowed to catch up (at a faster than listed rate). A larger number will result in larger spikes in rate but better average accuracy.
account_ratelimit	0	If set, will limit PUT and DELETE requests to /account_name/container_name. Number is in requests per second.
container_ratelimit_size		When set with container_ratelimit_x = r: for containers of size x, limit requests per second to r. Will limit PUT, DELETE, and POST requests to /a/c/o.
container_listing_ratelimit_size		When set with container_listing_ratelimit_x = r: for containers of size x, limit requests per second to r. Will limit GET requests to /a/c.

The container rate limits are linearly interpolated from the values given. A sample container rate limiting could be:

container_ratelimit_100 = 100

container_ratelimit_200 = 50

container_ratelimit_500 = 20

This would result in

Container Size	Rate Limit
0-99	No limiting
100	100
150	75
500	20
1000	20

2.9.2 Account Specific Ratelimiting

The above ratelimiting is to prevent the many writes to a single container bottleneck from causing a problem. There could also be a problem where a single account is just using too much of the clusters resources. In this case, the container ratelimits may not help because the customer could be doing thousands of reqs/sec to distributed containers each getting a small fraction of the total so those limits would never trigger. If a system administrator notices this, he/she can set the X-Account-Sysmeta-Global-Write-Ratelimit on an account and that will limit the total number of write requests (PUT, POST, DELETE,

COPY) that account can do for the whole account. This limit will be in addition to the applicable account/container limits from above. This header will be hidden from the user, because of the gatekeeper middleware, and can only be set using a direct client to the account nodes. It accepts a float value and will only limit requests if the value is > 0.

2.9.3 Black/White-listing

To blacklist or whitelist an account set:

X-Account-Sysmeta-Global-Write-Ratelimit: BLACKLIST

or

X-Account-Sysmeta-Global-Write-Ratelimit: WHITELIST

in the account headers.

2.10 Large Object Support

2.10.1 Overview

Swift has a limit on the size of a single uploaded object; by default this is 5GB. However, the download size of a single object is virtually unlimited with the concept of segmentation. Segments of the larger object are uploaded and a special manifest file is created that, when downloaded, sends all the segments concatenated as a single object. This also offers much greater upload speed with the possibility of parallel uploads of the segments.

2.10.2 Dynamic Large Objects

Middleware that will provide Dynamic Large Object (DLO) support.

Using `swift`

The quickest way to try out this feature is use the `swift` Swift Tool included with the python-swiftclient library. You can use the `-S` option to specify the segment size to use when splitting a large file. For example:

```
swift upload test_container -S 1073741824 large_file
```

This would split the large_file into 1G segments and begin uploading those segments in parallel. Once all the segments have been uploaded, `swift` will then create the manifest file so the segments can be downloaded as one.

So now, the following `swift` command would download the entire large object:

```
swift download test_container large_file
```

`swift` command uses a strict convention for its segmented object support. In the above example it will upload all the segments into a second container named test_container_segments. These segments will have names like large_file/1290206778.25/21474836480/00000000, large_file/1290206778.25/21474836480/00000001, etc.

The main benefit for using a separate container is that the main container listings will not be polluted with all the segment names. The reason for using the segment name format of <name>/<timestamp>/<size>/ is so that an upload of a new file with the same name wont overwrite the contents of the first until the last moment when the manifest file is updated.

`swift` will manage these segment files for you, deleting old segments on deletes and overwrites, etc. You can override this behavior with the `--leave-segments` option if desired; this is useful if you want to have multiple versions of the same large object available.

Direct API

You can also work with the segments and manifests directly with HTTP requests instead of having `swift` do that for you. You can just upload the segments like you would any other object and the manifest is just a zero-byte (not enforced) file with an extra `X-Object-Manifest` header.

All the object segments need to be in the same container, have a common object name prefix, and sort in the order in which they should be concatenated. Object names are sorted lexicographically as UTF-8 byte strings. They dont have to be in the same container as the manifest file will be, which is useful to keep container listings clean as explained above with `swift`.

The manifest file is simply a zero-byte (not enforced) file with the extra `X-Object-Manifest: <container>/<prefix>` header, where `<container>` is the container the object segments are in and `<prefix>` is the common prefix for all the segments.

It is best to upload all the segments first and then create or update the manifest. In this way, the full object wont be available for downloading until the upload is complete. Also, you can upload a new set of segments to a second location and then update the manifest to point to this new location. During the upload of the new segments, the original manifest will still be available to download the first set of segments.

Note: When updating a manifest object using a POST request, a `X-Object-Manifest` header must be included for the object to continue to behave as a manifest object.

The manifest file should have no content. However, this is not enforced. If the manifest path itself conforms to container/prefix specified in `X-Object-Manifest`, and if manifest has some content/data in it, it would also be considered as segment and manifests content will be part of the concatenated GET response. The order of concatenation follows the usual DLO logic which is - the order of concatenation adheres to order returned when segment names are sorted.

Heres an example using `curl` with tiny 1-byte segments:

```
# First, upload the segments
curl -X PUT -H 'X-Auth-Token: <token>'          http://<storage_url>/container/
↪myobject/00000001 --data-binary '1'
curl -X PUT -H 'X-Auth-Token: <token>'          http://<storage_url>/container/
↪myobject/00000002 --data-binary '2'
curl -X PUT -H 'X-Auth-Token: <token>'          http://<storage_url>/container/
↪myobject/00000003 --data-binary '3'

# Next, create the manifest file
curl -X PUT -H 'X-Auth-Token: <token>'          -H 'X-Object-Manifest:␣
↪container/myobject/'          http://<storage_url>/container/myobject --data-
↪binary ''
```

(continues on next page)

(continued from previous page)

```
# And now we can download the segments as a single object
curl -H 'X-Auth-Token: <token>'          http://<storage_url>/container/
↪myobject
```

class swift.common.middleware.dlo.**GetContext**(*dlo*, *logger*)

Bases: *swift.common.wsgi.WSGIContext*

get_or_head_response(*req*, *x_object_manifest*)

Parameters

- **req** users request
- **x_object_manifest** as unquoted, native string

handle_request(*req*, *start_response*)

Take a GET or HEAD request, and if it is for a dynamic large object manifest, return an appropriate response.

Otherwise, simply pass it through.

2.10.3 Static Large Objects

Middleware that will provide Static Large Object (SLO) support.

This feature is very similar to Dynamic Large Object (DLO) support in that it allows the user to upload many objects concurrently and afterwards download them as a single object. It is different in that it does not rely on eventually consistent container listings to do so. Instead, a user defined manifest of the object segments is used.

Uploading the Manifest

After the user has uploaded the objects to be concatenated, a manifest is uploaded. The request must be a PUT with the query parameter:

```
?multipart-manifest=put
```

The body of this request will be an ordered list of segment descriptions in JSON format. The data to be supplied for each segment is either:

Key	Description
path	the path to the segment object (not including account) /container/object_name
etag	(optional) the ETag given back when the segment object was PUT
size_bytes	(optional) the size of the complete segment object in bytes
range	(optional) the (inclusive) range within the object to use as a segment. If omitted, the entire object is used

Or:

Key	Description
data	base64-encoded data to be returned

Note: At least one object-backed segment must be included. If youd like to create a manifest consisting purely of data segments, consider uploading a normal object instead.

The format of the list will be:

```
[{"path": "/cont/object",
  "etag": "etagoftheobjectsegment",
  "size_bytes": 10485760,
  "range": "1048576-2097151"},
 {"data": base64.b64encode("interstitial data")},
 {"path": "/cont/another-object", ...},
 ...]
```

The number of object-backed segments is limited to `max_manifest_segments` (configurable in proxy-server.conf, default 1000). Each segment must be at least 1 byte. On upload, the middleware will head every object-backed segment passed in to verify:

1. the segment exists (i.e. the `HEAD` was successful);
2. the segment meets minimum size requirements;
3. if the user provided a non-null `etag`, the etag matches;
4. if the user provided a non-null `size_bytes`, the size_bytes matches; and
5. if the user provided a `range`, it is a singular, syntactically correct range that is satisfiable given the size of the object referenced.

For inlined data segments, the middleware verifies each is valid, non-empty base64-encoded binary data. Note that data segments *do not* count against `max_manifest_segments`.

Note that the `etag` and `size_bytes` keys are optional; if omitted, the verification is not performed. If any of the objects fail to verify (not found, size/etag mismatch, below minimum size, invalid range) then the user will receive a 4xx error response. If everything does match, the user will receive a 2xx response and the SLO object is ready for downloading.

Note that large manifests may take a long time to verify; historically, clients would need to use a long read timeout for the connection to give Swift enough time to send a final `201 Created` or `400 Bad Request` response. Now, clients should use the query parameters:

```
?multipart-manifest=put&heartbeat=on
```

to request that Swift send an immediate `202 Accepted` response and periodic whitespace to keep the connection alive. A final response code will appear in the body. The format of the response body defaults to text/plain but can be either json or xml depending on the `Accept` header. An example body is as follows:

```
Response Status: 201 Created
Response Body:
Etag: "8f481cede6d2ddc07cb36aa084d9a64d"
```

(continues on next page)

(continued from previous page)

```
Last Modified: Wed, 25 Oct 2017 17:08:55 GMT
Errors:
```

Or, as a json response:

```
{"Response Status": "201 Created",
 "Response Body": "",
 "Etag": "\"8f481cede6d2ddc07cb36aa084d9a64d\"",
 "Last Modified": "Wed, 25 Oct 2017 17:08:55 GMT",
 "Errors": []}
```

Behind the scenes, on success, a JSON manifest generated from the user input is sent to object servers with an extra `X-Static-Large-Object: True` header and a modified `Content-Type`. The items in this manifest will include the `etag` and `size_bytes` for each segment, regardless of whether the client specified them for verification. The parameter `swift_bytes=$total_size` will be appended to the existing `Content-Type`, where `$total_size` is the sum of all the included segments `size_bytes`. This extra parameter will be hidden from the user.

Manifest files can reference objects in separate containers, which will improve concurrent upload speed. Objects can be referenced by multiple manifests. The segments of a SLO manifest can even be other SLO manifests. Treat them as any other object i.e., use the `Etag` and `Content-Length` given on the `PUT` of the sub-SLO in the manifest to the parent SLO.

While uploading a manifest, a user can send `Etag` for verification. It needs to be md5 of the segments etags, if there is no range specified. For example, if the manifest to be uploaded looks like this:

```
[{"path": "/cont/object1",
  "etag": "etagoftheobjectsegment1",
  "size_bytes": 10485760},
 {"path": "/cont/object2",
  "etag": "etagoftheobjectsegment2",
  "size_bytes": 10485760}]
```

The Etag of the above manifest would be md5 of `etagoftheobjectsegment1` and `etagoftheobjectsegment2`. This could be computed in the following way:

```
echo -n 'etagoftheobjectsegment1etagoftheobjectsegment2' | md5sum
```

If a manifest to be uploaded with a segment range looks like this:

```
[{"path": "/cont/object1",
  "etag": "etagoftheobjectsegmentone",
  "size_bytes": 10485760,
  "range": "1-2"},
 {"path": "/cont/object2",
  "etag": "etagoftheobjectsegmenttwo",
  "size_bytes": 10485760,
  "range": "3-4"}]
```

While computing the Etag of the above manifest, internally each segments etag will be taken in the form of `etagvalue:rangevalue;`. Hence the Etag of the above manifest would be:

```
echo -n 'etagoftheobjectsegmentone:1-2;etagoftheobjectsegmenttwo:3-4;' \
| md5sum
```

For the purposes of Etag computations, inlined data segments are considered to have an etag of the md5 of the raw data (i.e., *not* base64-encoded).

Range Specification

Users now have the ability to specify ranges for SLO segments. Users can include an optional `range` field in segment descriptions to specify which bytes from the underlying object should be used for the segment data. Only one range may be specified per segment.

Note: The `etag` and `size_bytes` fields still describe the backing object as a whole.

If a user uploads this manifest:

```
[{"path": "/con/obj_seg_1", "size_bytes": 2097152, "range": "0-1048576"},
 {"path": "/con/obj_seg_2", "size_bytes": 2097152,
  "range": "512-1550000"},
 {"path": "/con/obj_seg_1", "size_bytes": 2097152, "range": "-2048"}]
```

The segment will consist of the first 1048576 bytes of /con/obj_seg_1, followed by bytes 513 through 1550000 (inclusive) of /con/obj_seg_2, and finally bytes 2095104 through 2097152 (i.e., the last 2048 bytes) of /con/obj_seg_1.

Note: The minimum sized range is 1 byte. This is the same as the minimum segment size.

Inline Data Specification

When uploading a manifest, users can include data segments that should be included along with objects. The data in these segments must be base64-encoded binary data and will be included in the etag of the resulting large object exactly as if that data had been uploaded and referenced as separate objects.

Note: This feature is primarily aimed at reducing the need for storing many tiny objects, and as such any supplied data must fit within the maximum manifest size (default is 8MiB). This maximum size can be configured via `max_manifest_size` in proxy-server.conf.

Retrieving a Large Object

A GET request to the manifest object will return the concatenation of the objects from the manifest much like DLO. If any of the segments from the manifest are not found or their Etag/Content-Length have changed since upload, the connection will drop. In this case a 409 Conflict will be logged in the proxy logs and the user will receive incomplete results. Note that this will be enforced regardless of whether the user performed per-segment validation during upload.

The headers from this GET or HEAD request will return the metadata attached to the manifest object itself with some exceptions:

Header	Value
Content-Length	the total size of the SLO (the sum of the sizes of the segments in the manifest)
X-Static-Large-Object	the string True
Etag	the etag of the SLO (generated the same way as DLO)

A GET request with the query parameter:

```
?multipart-manifest=get
```

will return a transformed version of the original manifest, containing additional fields and different key names. For example, the first manifest in the example above would look like this:

```
[{"name": "/cont/object",
  "hash": "etagoftheobjectsegment",
  "bytes": 10485760,
  "range": "1048576-2097151"}, ...]
```

As you can see, some of the fields are renamed compared to the put request: *path* is *name*, *etag* is *hash*, *size_bytes* is *bytes*. The *range* field remains the same (if present).

A GET request with the query parameters:

```
?multipart-manifest=get&format=raw
```

will return the contents of the original manifest as it was sent by the client. The main purpose for both calls is solely debugging.

When the manifest object is uploaded you are more or less guaranteed that every segment in the manifest exists and matched the specifications. However, there is nothing that prevents the user from breaking the SLO download by deleting/replacing a segment referenced in the manifest. It is left to the user to use caution in handling the segments.

Deleting a Large Object

A DELETE request will just delete the manifest object itself. The segment data referenced by the manifest will remain unchanged.

A DELETE with a query parameter:

```
?multipart-manifest=delete
```

will delete all the segments referenced in the manifest and then the manifest itself. The failure response will be similar to the bulk delete middleware.

A DELETE with the query parameters:

```
?multipart-manifest=delete&async=yes
```

will schedule all the segments referenced in the manifest to be deleted asynchronously and then delete the manifest itself. Note that segments will continue to appear in listings and be counted for quotas until they are cleaned up by the object-expirer. This option is only available when all segments are in the same container and none of them are nested SLOs.

Modifying a Large Object

PUT and POST requests will work as expected; PUTs will just overwrite the manifest object for example.

Container Listings

In a container listing the size listed for SLO manifest objects will be the `total_size` of the concatenated segments in the manifest. The overall `X-Container-Bytes-Used` for the container (and subsequently for the account) will not reflect `total_size` of the manifest but the actual size of the JSON data stored. The reason for this somewhat confusing discrepancy is we want the container listing to reflect the size of the manifest object when it is downloaded. We do not, however, want to count the bytes-used twice (for both the manifest and the segments its referring to) in the container and account metadata which can be used for stats and billing purposes.

class swift.common.middleware.slo.**SloGetContext**(*slo*)

Bases: *swift.common.wsgi.WSGIContext*

convert_segment_listing(*resp_headers*, *resp_iter*)

Converts the manifest data to match with the format that was put in through ?multipart-manifest=put

Parameters

- **resp_headers** response headers
- **resp_iter** a response iterable

handle_slo_get_or_head(*req*, *start_response*)

Takes a request and a start_response callable and does the normal WSGI thing with them. Returns an iterator suitable for sending up the WSGI chain.

Parameters

- **req** *Request* object; is a GET or HEAD request aimed at what may (or may not) be a static large object manifest.
- **start_response** WSGI start_response callable

class swift.common.middleware.slo.**StaticLargeObject**(*app*, *conf*, *max_manifest_segments=1000*, *max_manifest_size=8388608*, *yield_frequency=10*, *allow_async_delete=False*)

Bases: object

StaticLargeObject Middleware

See above for a full description.

The proxy logs created for any subrequests made will have swift.source set to SLO.

Parameters

- **app** The next WSGI filter or app in the paste.deploy chain.
- **conf** The configuration dict for the middleware.
- **max_manifest_segments** The maximum number of segments allowed in newly-created static large objects.
- **max_manifest_size** The maximum size (in bytes) of newly-created static-large-object manifests.
- **yield_frequency** If the client included heartbeat=on in the query parameters when creating a new static large object, the period of time to wait between sending whitespace to keep the connection alive.

get_segments_to_delete_iter(*req*)

A generator function to be used to delete all the segments and sub-segments referenced in a manifest.

Parameters **req** a *Request* with an SLO manifest in path

Raises

- **HTTPPreconditionFailed** on invalid UTF8 in request path
- **HTTPBadRequest** on too many buffered sub segments and on invalid SLO manifest path

get_slo_segments(*obj_name*, *req*)

Performs a *Request* and returns the SLO manifests segments.

Parameters

- **obj_name** the name of the object being deleted, as /container/object
- **req** the base *Request*

Raises

- **HTTPServerError** on unable to load obj_name or on unable to load the SLO manifest data.
- **HTTPBadRequest** on not an SLO manifest

- **HTTPNotFound** on SLO manifest not found

Returns SLO manifests segments

handle_multipart_delete(*req*)

Will delete all the segments in the SLO manifest and then, if successful, will delete the manifest file.

Parameters **req** a *Request* with an obj in path

Returns swob.Response whose app_iter set to Bulk.handle_delete_iter

handle_multipart_get_or_head(*req*, *start_response*)

Handles the GET or HEAD of a SLO manifest.

The response body (only on GET, of course) will consist of the concatenation of the segments.

Parameters

- **req** a *Request* with a path referencing an object
- **start_response** WSGI start_response callable

Raises **HttpException** on errors

handle_multipart_put(*req*, *start_response*)

Will handle the PUT of a SLO manifest. Heads every object in manifest to check if is valid and if so will save a manifest generated from the user input. Uses WSGIContext to call self and start_response and returns a WSGI iterator.

Parameters

- **req** a *Request* with an obj in path
- **start_response** WSGI start_response callable

Raises **HttpException** on errors

swift.common.middleware.slo.**parse_and_validate_input**(*req_body*, *req_path*)

Given a request body, parses it and returns a list of dictionaries.

The output structure is nearly the same as the input structure, but it is not an exact copy. Given a valid object-backed input dictionary `d_in`, its corresponding output dictionary `d_out` will be as follows:

- d_out[etag] == d_in[etag]
- d_out[path] == d_in[path]
- d_in[size_bytes] can be a string (12) or an integer (12), but d_out[size_bytes] is an integer.
- (optional) d_in[range] is a string of the form M-N, M-, or -N, where M and N are non-negative integers. d_out[range] is the corresponding swob.Range object. If d_in does not have a key range, neither will d_out.

Inlined data dictionaries will have any extraneous padding stripped.

Raises HTTPException on parse errors or semantic errors (e.g. bogus JSON structure, syntactically invalid ranges)

Returns a list of dictionaries on success

2.10.4 Direct API

SLO support centers around the user generated manifest file. After the user has uploaded the segments into their account a manifest file needs to be built and uploaded. All object segments, must be at least 1 byte in size. Please see the SLO docs for *Static Large Objects* further details.

2.10.5 Additional Notes

- With a `GET` or `HEAD` of a manifest file, the `X-Object-Manifest: <container>/<prefix>` header will be returned with the concatenated object so you can tell where its getting its segments from.
- When updating a manifest object using a POST request, a `X-Object-Manifest` header must be included for the object to continue to behave as a manifest object.
- The responses `Content-Length` for a `GET` or `HEAD` on the manifest file will be the sum of all the segments in the `<container>/<prefix>` listing, dynamically. So, uploading additional segments after the manifest is created will cause the concatenated object to be that much larger; theres no need to recreate the manifest file.
- The responses `Content-Type` for a `GET` or `HEAD` on the manifest will be the same as the `Content-Type` set during the `PUT` request that created the manifest. You can easily change the `Content-Type` by reissuing the `PUT`.
- The responses `ETag` for a `GET` or `HEAD` on the manifest file will be the MD5 sum of the concatenated string of ETags for each of the segments in the manifest (for DLO, from the listing `<container>/<prefix>`). Usually in Swift the ETag is the MD5 sum of the contents of the object, and that holds true for each segment independently. But its not meaningful to generate such an ETag for the manifest itself so this method was chosen to at least offer change detection.

Note: If you are using the container sync feature you will need to ensure both your manifest file and your segment files are synced if they happen to be in different containers.

2.10.6 History

Dynamic large object support has gone through various iterations before settling on this implementation.

The primary factor driving the limitation of object size in Swift is maintaining balance among the partitions of the ring. To maintain an even dispersion of disk usage throughout the cluster the obvious storage pattern was to simply split larger objects into smaller segments, which could then be glued together during a read.

Before the introduction of large object support some applications were already splitting their uploads into segments and re-assembling them on the client side after retrieving the individual pieces. This design allowed the client to support backup and archiving of large data sets, but was also frequently employed to improve performance or reduce errors due to network interruption. The major disadvantage of this method is that knowledge of the original partitioning scheme is required to properly reassemble the object, which is not practical for some use cases, such as CDN origination.

In order to eliminate any barrier to entry for clients wanting to store objects larger than 5GB, initially we also prototyped fully transparent support for large object uploads. A fully transparent implementation would support a larger max size by automatically splitting objects into segments during upload within

the proxy without any changes to the client API. All segments were completely hidden from the client API.

This solution introduced a number of challenging failure conditions into the cluster, wouldnt provide the client with any option to do parallel uploads, and had no basis for a resume feature. The transparent implementation was deemed just too complex for the benefit.

The current user manifest design was chosen in order to provide a transparent download of large objects to the client and still provide the uploading client a clean API to support segmented uploads.

To meet an many use cases as possible Swift supports two types of large object manifests. Dynamic and static large object manifests both support the same idea of allowing the user to upload many segments to be later downloaded as a single file.

Dynamic large objects rely on a container listing to provide the manifest. This has the advantage of allowing the user to add/removes segments from the manifest at any time. It has the disadvantage of relying on eventually consistent container listings. All three copies of the container dbs must be updated for a complete list to be guaranteed. Also, all segments must be in a single container, which can limit concurrent upload speed.

Static large objects rely on a user provided manifest file. A user can upload objects into multiple containers and then reference those objects (segments) in a self generated manifest file. Future GETs to that file will download the concatenation of the specified segments. This has the advantage of being able to immediately download the complete object once the manifest has been successfully PUT. Being able to upload segments into separate containers also improves concurrent upload speed. It has the disadvantage that the manifest is finalized once PUT. Any changes to it means it has to be replaced.

Between these two methods the user has great flexibility in how (s)he chooses to upload and retrieve large objects to Swift. Swift does not, however, stop the user from harming themselves. In both cases the segments are deletable by the user at any time. If a segment was deleted by mistake, a dynamic large object, having no way of knowing it was ever there, would happily ignore the deleted file and the user will get an incomplete file. A static large object would, when failing to retrieve the object specified in the manifest, drop the connection and the user would receive partial results.

2.11 Global Clusters

2.11.1 Overview

Swifts default configuration is currently designed to work in a single region, where a region is defined as a group of machines with high-bandwidth, low-latency links between them. However, configuration options exist that make running a performant multi-region Swift cluster possible.

For the rest of this section, we will assume a two-region Swift cluster: region 1 in San Francisco (SF), and region 2 in New York (NY). Each region shall contain within it 3 zones, numbered 1, 2, and 3, for a total of 6 zones.

2.11.2 Configuring Global Clusters

Note: The proxy-server configuration options described below can be given generic settings in the `[app:proxy-server]` configuration section and/or given specific settings for individual policies using *Per policy configuration*.

read_affinity

This setting, combined with sorting_method setting, makes the proxy server prefer local backend servers for GET and HEAD requests over non-local ones. For example, it is preferable for an SF proxy server to service object GET requests by talking to SF object servers, as the client will receive lower latency and higher throughput.

By default, Swift randomly chooses one of the three replicas to give to the client, thereby spreading the load evenly. In the case of a geographically-distributed cluster, the administrator is likely to prioritize keeping traffic local over even distribution of results. This is where the read_affinity setting comes in.

Example:

```
[app:proxy-server]
sorting_method = affinity
read_affinity = r1=100
```

This will make the proxy attempt to service GET and HEAD requests from backends in region 1 before contacting any backends in region 2. However, if no region 1 backends are available (due to replica placement, failed hardware, or other reasons), then the proxy will fall back to backend servers in other regions.

Example:

```
[app:proxy-server]
sorting_method = affinity
read_affinity = r1z1=100, r1=200
```

This will make the proxy attempt to service GET and HEAD requests from backends in region 1 zone 1, then backends in region 1, then any other backends. If a proxy is physically close to a particular zone or zones, this can provide bandwidth savings. For example, if a zone corresponds to servers in a particular rack, and the proxy server is in that same rack, then setting read_affinity to prefer reads from within the rack will result in less traffic between the top-of-rack switches.

The read_affinity setting may contain any number of region/zone specifiers; the priority number (after the equals sign) determines the ordering in which backend servers will be contacted. A lower number means higher priority.

Note that read_affinity only affects the ordering of primary nodes (see ring docs for definition of primary node), not the ordering of handoff nodes.

write_affinity

This setting makes the proxy server prefer local backend servers for object PUT requests over non-local ones. For example, it may be preferable for an SF proxy server to service object PUT requests by talking to SF object servers, as the client will receive lower latency and higher throughput. However, if this setting is used, note that a NY proxy server handling a GET request for an object that was PUT using write affinity may have to fetch it across the WAN link, as the object wont immediately have any replicas in NY. However, replication will move the objects replicas to their proper homes in both SF and NY.

One potential issue with write_affinity is, end user may get 404 error when deleting objects before replication. The write_affinity_handoff_delete_count setting is used together with write_affinity in order to solve that issue. With its default configuration, Swift will calculate the proper number of handoff nodes to send requests to.

Note that only object PUT/DELETE requests are affected by the write_affinity setting; POST, GET, HEAD, OPTIONS, and account/container PUT requests are not affected.

This setting lets you trade data distribution for throughput. If write_affinity is enabled, then object replicas will initially be stored all within a particular region or zone, thereby decreasing the quality of the data distribution, but the replicas will be distributed over fast WAN links, giving higher throughput to clients. Note that the replicators will eventually move objects to their proper, well-distributed homes.

The write_affinity setting is useful only when you dont typically read objects immediately after writing them. For example, consider a workload of mainly backups: if you have a bunch of machines in NY that periodically write backups to Swift, then odds are that you dont then immediately read those backups in SF. If your workload doesnt look like that, then you probably shouldnt use write_affinity.

The write_affinity_node_count setting is only useful in conjunction with write_affinity; it governs how many local object servers will be tried before falling back to non-local ones.

Example:

```
[app:proxy-server]
write_affinity = r1
write_affinity_node_count = 2 * replicas
```

Assuming 3 replicas, this configuration will make object PUTs try storing the objects replicas on up to 6 disks (2 * replicas) in region 1 (r1). Proxy server tries to find 3 devices for storing the object. While a device is unavailable, it queries the ring for the 4th device and so on until 6th device. If the 6th disk is still unavailable, the last replica will be sent to other region. It doesnt mean therell have 6 replicas in region 1.

You should be aware that, if you have data coming into SF faster than your replicators are transferring it to NY, then your clusters data distribution will get worse and worse over time as objects pile up in SF. If this happens, it is recommended to disable write_affinity and simply let object PUTs traverse the WAN link, as that will naturally limit the object growth rate to what your WAN link can handle.

2.12 Container to Container Synchronization

2.12.1 Overview

Swift has a feature where all the contents of a container can be mirrored to another container through background synchronization. Swift cluster operators configure their cluster to allow/accept sync requests to/from other clusters, and the user specifies where to sync their container to along with a secret synchronization key.

Note: If you are using the *Large Objects* feature and syncing to another cluster then you will need to ensure that manifest files and segment files are synced. If segment files are in a different container than their manifest then both the manifests container and the segments container must be synced. The target container for synced segment files must always have the same name as their source container in order for them to be resolved by synced manifests.

Be aware that manifest files may be synced before segment files even if they are in the same container and were created after the segment files.

In the case of *Static Large Objects*, a GET request for a manifest whose segments have yet to be completely synced will fail with none or only part of the large object content being returned.

In the case of *Dynamic Large Objects*, a GET request for a manifest whose segments have yet to be completely synced will either fail or return unexpected (and most likely incorrect) content.

Note: If you are using encryption middleware in the cluster from which objects are being synced, then you should follow the instructions for *Container sync configuration* to be compatible with encryption.

Note: If you are using symlink middleware in the cluster from which objects are being synced, then you should follow the instructions for *Container sync configuration* to be compatible with symlinks.

Be aware that symlinks may be synced before their targets even if they are in the same container and were created after the target objects. In such cases, a GET for the symlink will fail with a `404 Not Found` error. If the target has been overwritten, a GET may produce an older version (for dynamic links) or a `409 Conflict` error (for static links).

2.12.2 Configuring Container Sync

Create a `container-sync-realms.conf` file specifying the allowable clusters and their information:

```
[realm1]
key = realm1key
key2 = realm1key2
cluster_clustername1 = https://host1/v1/
cluster_clustername2 = https://host2/v1/

[realm2]
key = realm2key
```

(continues on next page)

(continued from previous page)

```
key2 = realm2key2
cluster_clustername3 = https://host3/v1/
cluster_clustername4 = https://host4/v1/
```

Each section name is the name of a sync realm. A sync realm is a set of clusters that have agreed to allow container syncing with each other. Realm names will be considered case insensitive.

`key` is the overall cluster-to-cluster key used in combination with the external users key that they set on their containers `X-Container-Sync-Key` metadata header values. These keys will be used to sign each request the container sync daemon makes and used to validate each incoming container sync request.

`key2` is optional and is an additional key incoming requests will be checked against. This is so you can rotate keys if you wish; you move the existing `key` to `key2` and make a new `key` value.

Any values in the realm section whose names begin with `cluster_` will indicate the name and endpoint of a cluster and will be used by external users in their containers `X-Container-Sync-To` metadata header values with the format `//realm_name/cluster_name/account_name/container_name`. Realm and cluster names are considered case insensitive.

The endpoint is what the container sync daemon will use when sending out requests to that cluster. Keep in mind this endpoint must be reachable by all container servers, since that is where the container sync daemon runs. Note that the endpoint ends with `/v1/` and that the container sync daemon will then add the `account/container/obj` name after that.

Distribute this `container-sync-realms.conf` file to all your proxy servers and container servers.

You also need to add the container_sync middleware to your proxy pipeline. It needs to be after any memcache middleware and before any auth middleware. The `[filter:container_sync]` section only needs the `use` item. For example:

```
[pipeline:main]
pipeline = healthcheck proxy-logging cache container_sync tempauth proxy-
↪logging proxy-server

[filter:container_sync]
use = egg:swift#container_sync
```

The container sync daemon will use an internal client to sync objects. Even if you dont configure the internal client, the container sync daemon will work with default configuration. The default configuration is the same as `internal-client.conf-sample`. If you want to configure the internal client, please update `internal_client_conf_path` in `container-server.conf`. The configuration file at the path will be used for the internal client.

2.12.3 Old-Style: Configuring a Clusters Allowable Sync Hosts

This section is for the old-style of using container sync. See the previous section, Configuring Container Sync, for the new-style.

With the old-style, the Swift cluster operator must allow synchronization with a set of hosts before the user can enable container synchronization. First, the backend container server needs to be given this list of hosts in the `container-server.conf` file:

```
[DEFAULT]
# This is a comma separated list of hosts allowed in the
# X-Container-Sync-To field for containers.
# allowed_sync_hosts = 127.0.0.1
allowed_sync_hosts = host1,host2,etc.
...

[container-sync]
# You can override the default log routing for this app here (don't
# use set!):
# log_name = container-sync
# log_facility = LOG_LOCAL0
# log_level = INFO
# Will sync, at most, each container once per interval
# interval = 300
# Maximum amount of time to spend syncing each container
# container_time = 60
```

2.12.4 Logging Container Sync

Currently, log processing is the only way to track sync progress, problems, and even just general activity for container synchronization. In that light, you may wish to set the above `log_` options to direct the container-sync logs to a different file for easier monitoring. Additionally, it should be noted there is no way for an end user to monitor sync progress or detect problems other than HEADing both containers and comparing the overall information.

2.12.5 Container Sync Statistics

Container Sync INFO level logs contain activity metrics and accounting information for insightful tracking. Currently two different statistics are collected:

About once an hour or so, accumulated statistics of all operations performed by Container Sync are reported to the log file with the following format:

```
Since (time): (sync) synced [(delete) deletes, (put) puts], (skip) skipped,␣
↪(fail) failed
```

time last report time

sync number of containers with sync turned on that were successfully synced

delete number of successful DELETE object requests to the target cluster

put number of successful PUT object request to the target cluster

skip number of containers whose sync has been turned off, but are not yet cleared from the sync store

fail number of containers with failure (due to exception, timeout or other reason)

For each container synced, per container statistics are reported with the following format:

```
Container sync report: (container), time window start: (start), time window␣
↪end: %(end), puts: (puts), posts: (posts), deletes: (deletes), bytes:␣
↪(bytes), sync_point1: (point1), sync_point2: (point2), total_rows: (total)
```

(continues on next page)

(continued from previous page)

container account/container statistics are for

start report start time

end report end time

puts number of successful PUT object requests to the target container

posts N/A (0)

deletes number of successful DELETE object requests to the target container

bytes number of bytes sent over the network to the target container

point1 progress indication - the containers `x_container_sync_point1`

point2 progress indication - the containers `x_container_sync_point2`

total number of objects processed at the container

It is possible that more than one server syncs a container, therefore log files from all servers need to be evaluated

2.12.6 Using the `swift` tool to set up synchronized containers

Note: The `swift` tool is available from the python-swiftclient library.

Note: You must be the account admin on the account to set synchronization targets and keys.

You simply tell each container where to sync to and give it a secret synchronization key. First, lets get the account details for our two cluster accounts:

```
$ swift -A http://cluster1/auth/v1.0 -U test:tester -K testing stat -v
StorageURL: http://cluster1/v1/AUTH_208d1854-e475-4500-b315-81de645d060e
Auth Token: AUTH_tkd5359e46ff9e419fa193dbd367f3cd19
   Account: AUTH_208d1854-e475-4500-b315-81de645d060e
Containers: 0
   Objects: 0
     Bytes: 0

$ swift -A http://cluster2/auth/v1.0 -U test2:tester2 -K testing2 stat -v
StorageURL: http://cluster2/v1/AUTH_33cdcad8-09fb-4940-90da-0f00cbf21c7c
Auth Token: AUTH_tk816a1aaf403c49adb92ecfca2f88e430
   Account: AUTH_33cdcad8-09fb-4940-90da-0f00cbf21c7c
Containers: 0
   Objects: 0
     Bytes: 0
```

Now, lets make our first container and tell it to synchronize to a second well make next:

```
$ swift -A http://cluster1/auth/v1.0 -U test:tester -K testing post \
  -t '//realm_name/clustername2/AUTH_33cdcad8-09fb-4940-90da-0f00cbf21c7c/
↪container2' \
  -k 'secret' container1
```

The `-t` indicates the cluster to sync to, which is the realm name of the section from `container-sync-realms.conf`, followed by the cluster name from that section (without the `cluster_` prefix), followed by the account and container names we want to sync to. The `-k` specifies the secret key the two containers will share for synchronization; this is the user key, the cluster key in `container-sync-realms.conf` will also be used behind the scenes.

Now, well do something similar for the second clusters container:

```
$ swift -A http://cluster2/auth/v1.0 -U test2:tester2 -K testing2 post \
  -t '//realm_name/clustername1/AUTH_208d1854-e475-4500-b315-81de645d060e/
↪container1' \
  -k 'secret' container2
```

Thats it. Now we can upload a bunch of stuff to the first container and watch as it gets synchronized over to the second:

```
$ swift -A http://cluster1/auth/v1.0 -U test:tester -K testing \
  upload container1 .
photo002.png
photo004.png
photo001.png
photo003.png

$ swift -A http://cluster2/auth/v1.0 -U test2:tester2 -K testing2 \
  list container2

[Nothing there yet, so we wait a bit...]
```

Note: If youre an operator running *SAIO (Swift All In One)* and just testing, each time you configure a container for synchronization and place objects in the source container you will need to ensure that container-sync runs before attempting to retrieve objects from the target container. That is, you need to run:

```
swift-init container-sync once
```

Now expect to see objects copied from the first container to the second:

```
$ swift -A http://cluster2/auth/v1.0 -U test2:tester2 -K testing2 \
  list container2
photo001.png
photo002.png
photo003.png
photo004.png
```

You can also set up a chain of synced containers if you want more than two. Youd point 1 -> 2, then 2 ->

3, and finally 3 -> 1 for three containers. Theyd all need to share the same secret synchronization key.

2.12.7 Using curl (or other tools) instead

So whats `swift` doing behind the scenes? Nothing overly complicated. It translates the `-t <value>` option into an `X-Container-Sync-To: <value>` header and the `-k <value>` option into an `X-Container-Sync-Key: <value>` header.

For instance, when we created the first container above and told it to synchronize to the second, we could have used this curl command:

```
$ curl -i -X POST -H 'X-Auth-Token: AUTH_tkd5359e46ff9e419fa193dbd367f3cd19' \
  -H 'X-Container-Sync-To: //realm_name/clustername2/AUTH_33cdcad8-09fb-4940-
↪90da-0f00cbf21c7c/container2' \
  -H 'X-Container-Sync-Key: secret' \
  'http://cluster1/v1/AUTH_208d1854-e475-4500-b315-81de645d060e/container1'
HTTP/1.1 204 No Content
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
Date: Thu, 24 Feb 2011 22:39:14 GMT
```

2.12.8 Old-Style: Using the `swift` tool to set up synchronized containers

Note: The `swift` tool is available from the python-swiftclient library.

Note: You must be the account admin on the account to set synchronization targets and keys.

This is for the old-style of container syncing using `allowed_sync_hosts`.

You simply tell each container where to sync to and give it a secret synchronization key. First, lets get the account details for our two cluster accounts:

```
$ swift -A http://cluster1/auth/v1.0 -U test:tester -K testing stat -v
StorageURL: http://cluster1/v1/AUTH_208d1854-e475-4500-b315-81de645d060e
Auth Token: AUTH_tkd5359e46ff9e419fa193dbd367f3cd19
   Account: AUTH_208d1854-e475-4500-b315-81de645d060e
Containers: 0
   Objects: 0
     Bytes: 0

$ swift -A http://cluster2/auth/v1.0 -U test2:tester2 -K testing2 stat -v
StorageURL: http://cluster2/v1/AUTH_33cdcad8-09fb-4940-90da-0f00cbf21c7c
Auth Token: AUTH_tk816a1aaf403c49adb92ecfca2f88e430
   Account: AUTH_33cdcad8-09fb-4940-90da-0f00cbf21c7c
Containers: 0
   Objects: 0
     Bytes: 0
```

Now, lets make our first container and tell it to synchronize to a second well make next:

```
$ swift -A http://cluster1/auth/v1.0 -U test:tester -K testing post \
  -t 'http://cluster2/v1/AUTH_33cdcad8-09fb-4940-90da-0f00cbf21c7c/container2
↪' \
  -k 'secret' container1
```

The `-t` indicates the URL to sync to, which is the `StorageURL` from cluster2 we retrieved above plus the container name. The `-k` specifies the secret key the two containers will share for synchronization. Now, well do something similar for the second clusters container:

```
$ swift -A http://cluster2/auth/v1.0 -U test2:tester2 -K testing2 post \
  -t 'http://cluster1/v1/AUTH_208d1854-e475-4500-b315-81de645d060e/container1
↪' \
  -k 'secret' container2
```

Thats it. Now we can upload a bunch of stuff to the first container and watch as it gets synchronized over to the second:

```
$ swift -A http://cluster1/auth/v1.0 -U test:tester -K testing \
  upload container1 .
photo002.png
photo004.png
photo001.png
photo003.png

$ swift -A http://cluster2/auth/v1.0 -U test2:tester2 -K testing2 \
  list container2

[Nothing there yet, so we wait a bit...]
[If you're an operator running SAIO and just testing, you may need to
 run 'swift-init container-sync once' to perform a sync scan.]

$ swift -A http://cluster2/auth/v1.0 -U test2:tester2 -K testing2 \
  list container2
photo001.png
photo002.png
photo003.png
photo004.png
```

You can also set up a chain of synced containers if you want more than two. Youd point 1 -> 2, then 2 -> 3, and finally 3 -> 1 for three containers. Theyd all need to share the same secret synchronization key.

2.12.9 Old-Style: Using curl (or other tools) instead

This is for the old-style of container syncing using `allowed_sync_hosts`.

So whats `swift` doing behind the scenes? Nothing overly complicated. It translates the `-t <value>` option into an `X-Container-Sync-To: <value>` header and the `-k <value>` option into an `X-Container-Sync-Key: <value>` header.

For instance, when we created the first container above and told it to synchronize to the second, we could have used this curl command:

```
$ curl -i -X POST -H 'X-Auth-Token: AUTH_tkd5359e46ff9e419fa193dbd367f3cd19' \
  -H 'X-Container-Sync-To: http://cluster2/v1/AUTH_33cdcad8-09fb-4940-90da-
↪0f00cbf21c7c/container2' \
  -H 'X-Container-Sync-Key: secret' \
  'http://cluster1/v1/AUTH_208d1854-e475-4500-b315-81de645d060e/container1'
HTTP/1.1 204 No Content
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
Date: Thu, 24 Feb 2011 22:39:14 GMT
```

2.12.10 Whats going on behind the scenes, in the cluster?

Container ring devices have a directory called `containers`, where container databases reside. In addition to `containers`, each container ring device also has a directory called `sync-containers`. `sync-containers` holds symlinks to container databases that were configured for container sync using `x-container-sync-to` and `x-container-sync-key` metadata keys.

The swift-container-sync process does the job of sending updates to the remote container. This is done by scanning `sync-containers` for container databases. For each container db found, newer rows since the last sync will trigger PUTs or DELETEs to the other container.

`sync-containers` is maintained as follows: Whenever the container-server processes a PUT or a POST request that carries `x-container-sync-to` and `x-container-sync-key` metadata keys the server creates a symlink to the container database in `sync-containers`. Whenever the container server deletes a synced container, the appropriate symlink is deleted from `sync-containers`.

In addition to the container-server, the container-replicator process does the job of identifying containers that should be synchronized. This is done by scanning the local devices for container databases and checking for `x-container-sync-to` and `x-container-sync-key` metadata values. If they exist then a symlink to the container database is created in a `sync-containers` sub-directory on the same device.

Similarly, when the container sync metadata keys are deleted, the container server and container-replicator would take care of deleting the symlinks from `sync-containers`.

Note: The swift-container-sync process runs on each container server in the cluster and talks to the proxy servers (or load balancers) in the remote cluster. Therefore, the container servers must be permitted to initiate outbound connections to the remote proxy servers (or load balancers).

The actual syncing is slightly more complicated to make use of the three (or number-of-replicas) main nodes for a container without each trying to do the exact same work but also without missing work if one node happens to be down.

Two sync points are kept in each container database. When syncing a container, the container-sync process figures out which replica of the container it has. In a standard 3-replica scenario, the process will have either replica number 0, 1, or 2. This is used to figure out which rows belong to this sync process and which ones dont.

An example may help. Assume a replica count of 3 and database row IDs are 1..6. Also, assume that container-sync is running on this container for the first time, hence SP1 = SP2 = -1.

```
SP1
SP2
 |
 v
-1 0 1 2 3 4 5 6
```

First, the container-sync process looks for rows with id between SP1 and SP2. Since this is the first run, SP1 = SP2 = -1, and there arent any such rows.

```
SP1
SP2
 |
 v
-1 0 1 2 3 4 5 6
```

Second, the container-sync process looks for rows with id greater than SP1, and syncs those rows which it owns. Ownership is based on the hash of the object name, so its not always guaranteed to be exactly one out of every three rows, but it usually gets close. For the sake of example, lets say that this process ends up owning rows 2 and 5.

Once its finished trying to sync those rows, it updates SP1 to be the biggest row-id that its seen, which is 6 in this example.

```
SP2            SP1
 |              |
 v              v
-1 0 1 2 3 4 5 6
```

While all that was going on, clients uploaded new objects into the container, creating new rows in the database.

```
SP2            SP1
 |              |
 v              v
-1 0 1 2 3 4 5 6 7 8 9 10 11 12
```

On the next run, the container-sync starts off looking at rows with ids between SP1 and SP2. This time, there are a bunch of them. The sync process try to sync all of them. If it succeeds, it will set SP2 to equal SP1. If it fails, it will set SP2 to the failed object and will continue to try all other objects till SP1, setting SP2 to the first object that failed.

Under normal circumstances, the container-sync processes will have already taken care of synchronizing all rows, between SP1 and SP2, resulting in a set of quick checks. However, if one of the sync processes failed for some reason, then this is a vital fallback to make sure all the objects in the container get synchronized. Without this seemingly-redundant work, any container-sync failure results in unsynchronized

objects. Note that the container sync will persistently retry to sync any faulty object until success, while logging each failure.

Once its done with the fallback rows, and assuming no faults occurred, SP2 is advanced to SP1.

Then, rows with row ID greater than SP1 are synchronized (provided this container-sync process is responsible for them), and SP1 is moved up to the greatest row ID seen.

2.13 Expiring Object Support

The `swift-object-expirer` offers scheduled deletion of objects. The Swift client would use the `X-Delete-At` or `X-Delete-After` headers during an object `PUT` or `POST` and the cluster would automatically quit serving that object at the specified time and would shortly thereafter remove the object from the system.

The `X-Delete-At` header takes a Unix Epoch timestamp, in integer form; for example: `1317070737` represents `Mon Sep 26 20:58:57 2011 UTC`.

The `X-Delete-After` header takes a positive integer number of seconds. The proxy server that receives the request will convert this header into an `X-Delete-At` header using the request timestamp plus the value given.

If both the `X-Delete-At` and `X-Delete-After` headers are sent with a request then the `X-Delete-After` header will take precedence.

As expiring objects are added to the system, the object servers will record the expirations in a hidden `.expiring_objects` account for the `swift-object-expirer` to handle later.

Usually, just one instance of the `swift-object-expirer` daemon needs to run for a cluster. This isnt exactly automatic failover high availability, but if this daemon doesnt run for a few hours it should not be any real issue. The expired-but-not-yet-deleted objects will still `404 Not Found` if someone tries to `GET` or `HEAD` them and theyll just be deleted a bit later when the daemon is restarted.

By default, the `swift-object-expirer` daemon will run with a concurrency of 1. Increase this value to get more concurrency. A concurrency of 1 may not be enough to delete expiring objects in a timely fashion for a particular Swift cluster.

It is possible to run multiple daemons to do different parts of the work if a single process with a concurrency of more than 1 is not enough (see the sample config file for details).

To run the `swift-object-expirer` as multiple processes, set `processes` to the number of processes (either in the config file or on the command line). Then run one process for each part. Use `process` to specify the part of the work to be done by a process using the command line or the config. So, for

example, if youd like to run three processes, set `processes` to 3 and run three processes with `process` set to 0, 1, and 2 for the three processes. If multiple processes are used, its necessary to run one for each part of the work or that part of the work will not be done.

By default the daemon looks for two different config files. When launching, the process searches for the `[object-expirer]` section in the

`/etc/swift/object-server.conf` config. If the section or the config is missing it will then look for and use the `/etc/swift/object-expirer.conf` config. The latter config file is considered deprecated and is searched for to aid in cluster upgrades.

2.13.1 Upgrading impact: General Task Queue vs Legacy Queue

The expirer daemon will be moving to a new general task-queue based design that will divide the work across all object servers, as such only expirers defined in the object-server config will be able to use the new system. The parameters in both files are identical except for a new option in the object-server `[object-expirer]` section, `dequeue_from_legacy` which when set to `True` will tell the expirer that in addition to using the new task queueing system to also check the legacy (soon to be deprecated) queue.

Note: The new task-queue system has not been completed yet. So an expirers with `dequeue_from_legacy` set to `False` will currently do nothing.

By default `dequeue_from_legacy` will be `False`, it is necessary to be set to `True` explicitly while migrating from the old expiring queue.

Any expirer using the old config `/etc/swift/object-expirer.conf` will not use the new general task queue. Itll ignore the `dequeue_from_legacy` and will only check the legacy queue. Meaning itll run as a legacy expirer.

Why is this important? If you are currently running object-expirers on nodes that are not object storage nodes, then for the time being they will still work but only by dequeuing from the old queue. When the new general task queue is introduced, expirers will be required to run on the object servers so that any new objects added can be removed. If youre in this situation, you can safely setup the new expirer section in the `object-server.conf` to deal with the new queue and leave the legacy expirers running elsewhere.

However, if your old expirers are running on the object-servers, the most common topology, then you would add the new section to all object servers, to deal the new queue. In order to maintain the same number of expirers checking the legacy queue, pick the same number of nodes as you previously had and turn on `dequeue_from_legacy` on those nodes only. Also note on these nodes youd need to keep the legacy `process` and `processes` options to maintain the concurrency level for the legacy queue.

Note: Be careful not to enable `dequeue_from_legacy` on too many expirers as all legacy tasks are stored in a single hidden account and the same hidden containers. On a large cluster one may inadvertently overload the acccount/container servers handling the legacy expirer queue.

Here is a quick sample of the `object-expirer` section required in the `object-server.conf`:

```
[object-expirer]
# log_name = object-expirer
# log_facility = LOG_LOCAL0
```

(continues on next page)

(continued from previous page)

```
# log_level = INFO
# log_address = /dev/log
#
interval = 300

# If this true, expirer execute tasks in legacy expirer task queue
dequeue_from_legacy = false

# processes can only be used in conjunction with `dequeue_from_legacy`.
# So this option is ignored if dequeue_from_legacy=false.
# processes is how many parts to divide the legacy work into, one part per
# process that will be doing the work
# processes set 0 means that a single legacy process will be doing all the␣
↪work
# processes can also be specified on the command line and will override the
# config value
# processes = 0

# process can only be used in conjunction with `dequeue_from_legacy`.
# So this option is ignored if dequeue_from_legacy=false.
# process is which of the parts a particular legacy process will work on
# process can also be specified on the command line and will override the␣
↪config
# value
# process is "zero based", if you want to use 3 processes, you should run
# processes with process set to 0, 1, and 2
# process = 0

report_interval = 300

# request_tries is the number of times the expirer's internal client will
# attempt any given request in the event of failure. The default is 3.
# request_tries = 3

# concurrency is the level of concurrency to use to do the work, this value
# must be set to at least 1
# concurrency = 1

# The expirer will re-attempt expiring if the source object is not available
# up to reclaim_age seconds before it gives up and deletes the entry in the
# queue.
# reclaim_age = 604800
```

And for completeness, here is a quick sample of the legacy `object-expirer.conf` file:

```
[DEFAULT]
# swift_dir = /etc/swift
# user = swift
# You can specify default log routing here if you want:
```

(continues on next page)

(continued from previous page)

```
# log_name = swift
# log_facility = LOG_LOCAL0
# log_level = INFO

[object-expirer]
interval = 300

[pipeline:main]
pipeline = catch_errors cache proxy-server

[app:proxy-server]
use = egg:swift#proxy
# See proxy-server.conf-sample for options

[filter:cache]
use = egg:swift#memcache
# See proxy-server.conf-sample for options

[filter:catch_errors]
use = egg:swift#catch_errors
# See proxy-server.conf-sample for options
```

Note: When running legacy expirers, the daemon needs to run on a machine with access to all the backend servers in the cluster, but does not need proxy server or public access. The daemon will use its own internal proxy code instance to access the backend servers.

2.14 CORS

CORS is a mechanism to allow code running in a browser (Javascript for example) make requests to a domain other than the one from where it originated.

Swift supports CORS requests to containers and objects.

CORS metadata is held on the container only. The values given apply to the container itself and all objects within it.

The supported headers are,

Metadata	Use
X-Container-Meta-Access-Control-Allow-Origin	Origins to be allowed to make Cross Origin Requests, space separated.
X-Container-Meta-Access-Control-Max-Age	Max age for the Origin to hold the preflight results.
X-Container-Meta-Access-Control-Expose-Headers	Headers exposed to the user agent (e.g. browser) in the actual request response. Space separated.

In addition the values set in container metadata, some cluster-wide values may also be configured using the `strict_cors_mode`, `cors_allow_origin` and `cors_expose_headers` in `proxy-server.conf`.

See `proxy-server.conf-sample` for more information.

Before a browser issues an actual request it may issue a preflight request. The preflight request is an OPTIONS call to verify the Origin is allowed to make the request. The sequence of events are,

- Browser makes OPTIONS request to Swift
- Swift returns 200/401 to browser based on allowed origins
- If 200, browser makes the actual request to Swift, i.e. PUT, POST, DELETE, HEAD, GET

When a browser receives a response to an actual request it only exposes those headers listed in the `Access-Control-Expose-Headers` header. By default Swift returns the following values for this header,

- simple response headers as listed on http://www.w3.org/TR/cors/#simple-response-header
- the headers `etag`, `x-timestamp`, `x-trans-id`, `x-openstack-request-id`
- all metadata headers (`X-Container-Meta-*` for containers and `X-Object-Meta-*` for objects)
- headers listed in `X-Container-Meta-Access-Control-Expose-Headers`
- headers configured using the `cors_expose_headers` option in `proxy-server.conf`

Note: An OPTIONS request to a symlink object will respond with the options for the symlink only, the request will not be redirected to the target object. Therefore, if the symlinks target object is in another container with CORS settings, the response will not reflect the settings.

2.14.1 Sample Javascript

To see some CORS Javascript in action download the *test CORS page* (source below). Host it on a webserver and take note of the protocol and hostname (origin) youll be using to request the page, e.g. http://localhost.

Locate a container youd like to query. Needless to say the Swift cluster hosting this container should have CORS support. Append the origin of the test page to the containers `X-Container-Meta-Access-Control-Allow-Origin` header,:

```
curl -X POST -H 'X-Auth-Token: xxx' \
  -H 'X-Container-Meta-Access-Control-Allow-Origin: http://localhost' \
  http://192.168.56.3:8080/v1/AUTH_test/cont1
```

At this point the container is now accessible to CORS clients hosted on http://localhost. Open the test CORS page in your browser.

1. Populate the Token field
2. Populate the URL field with the URL of either a container or object
3. Select the request method
4. Hit Submit

Assuming the request succeeds you should see the response header and body. If something went wrong the response status will be 0.

2.14.2 Test CORS Page

A sample cross-site test page is located in the project source tree `doc/source/test-cors.html`.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Test CORS</title>
  </head>
  <body>

    Token<br><input id="token" type="text" size="64"><br><br>

    Method<br>
    <select id="method">
        <option value="GET">GET</option>
        <option value="HEAD">HEAD</option>
        <option value="POST">POST</option>
        <option value="DELETE">DELETE</option>
        <option value="PUT">PUT</option>
    </select><br><br>

    URL (Container or Object)<br><input id="url" size="64" type="text"><br>
↪<br>

    <input id="submit" type="button" value="Submit" onclick="submit(); return␣
↪false;">

    <pre id="response_headers"></pre>
    <p>
    <hr>
    <pre id="response_body"></pre>

    <script type="text/javascript">
      function submit() {
          var token = document.getElementById('token').value;
          var method = document.getElementById('method').value;
          var url = document.getElementById('url').value;

          document.getElementById('response_headers').textContent = null;
          document.getElementById('response_body').textContent = null;

          var request = new XMLHttpRequest();

          request.onreadystatechange = function (oEvent) {
              if (request.readyState == 4) {
                  responseHeaders = 'Status: ' + request.status;
                  responseHeaders = responseHeaders + '\nStatus Text: ' +␣
↪request.statusText;
                  responseHeaders = responseHeaders + '\n\n' + request.
↪getAllResponseHeaders();
```

(continues on next page)

(continued from previous page)

```
                document.getElementById('response_headers').textContent =␣
↪responseHeaders;
                document.getElementById('response_body').textContent =␣
↪request.responseText;
            }
        }

        request.open(method, url);
        if (token != '') {
            // custom headers always trigger a pre-flight request
            request.setRequestHeader('X-Auth-Token', token);
        }
        request.send(null);
    }
  </script>

 </body>
</html>
```

2.15 Cross-domain Policy File

A cross-domain policy file allows web pages hosted elsewhere to use client side technologies such as Flash, Java and Silverlight to interact with the Swift API.

See http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html for a description of the purpose and structure of the cross-domain policy file. The cross-domain policy file is installed in the root of a web server (i.e., the path is /crossdomain.xml).

The crossdomain middleware responds to a path of /crossdomain.xml with an XML document such as:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM "http://www.adobe.com/xml/dtds/cross-
↪domain-policy.dtd" >
<cross-domain-policy>
    <allow-access-from domain="*" secure="false" />
</cross-domain-policy>
```

You should use a policy appropriate to your site. The examples and the default policy are provided to indicate how to syntactically construct a cross domain policy file they are not recommendations.

2.15.1 Configuration

To enable this middleware, add it to the pipeline in your proxy-server.conf file. It should be added before any authentication (e.g., tempauth or keystone) middleware. In this example ellipsis () indicate other middleware you may have chosen to use:

```
[pipeline:main]
pipeline =  ... crossdomain ... authtoken ... proxy-server
```

And add a filter section, such as:

```
[filter:crossdomain]
use = egg:swift#crossdomain
cross_domain_policy = <allow-access-from domain="*.example.com" />
    <allow-access-from domain="www.example.com" secure="false" />
```

For continuation lines, put some whitespace before the continuation text. Ensure you put a completely blank line to terminate the cross_domain_policy value.

The cross_domain_policy name/value is optional. If omitted, the policy defaults as if you had specified:

```
cross_domain_policy = <allow-access-from domain="*" secure="false" />
```

2.16 Erasure Code Support

2.16.1 History and Theory of Operation

Theres a lot of good material out there on Erasure Code (EC) theory, this short introduction is just meant to provide some basic context to help the reader better understand the implementation in Swift.

Erasure Coding for storage applications grew out of Coding Theory as far back as the 1960s with the Reed-Solomon codes. These codes have been used for years in applications ranging from CDs to DVDs to general communications and, yes, even in the space program starting with Voyager! The basic idea is that some amount of data is broken up into smaller pieces called fragments and coded in such a way that it can be transmitted with the ability to tolerate the loss of some number of the coded fragments. Thats where the word erasure comes in, if you transmit 14 fragments and only 13 are received then one of them is said to be erased. The word erasure provides an important distinction with EC; it isnt about detecting errors, its about dealing with failures. Another important element of EC is that the number of erasures that can be tolerated can be adjusted to meet the needs of the application.

At a high level EC works by using a specific scheme to break up a single data buffer into several smaller data buffers then, depending on the scheme, performing some encoding operation on that data in order to generate additional information. So you end up with more data than you started with and that extra data is often called parity. Note that there are many, many different encoding techniques that vary both in how they organize and manipulate the data as well by what means they use to calculate parity. For example, one scheme might rely on Galois Field Arithmetic while others may work with only XOR. The number of variations and details about their differences are well beyond the scope of this introduction, but we will talk more about a few of them when we get into the implementation of EC in Swift.

Overview of EC Support in Swift

First and foremost, from an application perspective EC support is totally transparent. There are no EC related external API; a container is simply created using a Storage Policy defined to use EC and then interaction with the cluster is the same as any other durability policy.

EC is implemented in Swift as a Storage Policy, see *Storage Policies* for complete details on Storage Policies. Because support is implemented as a Storage Policy, all of the storage devices associated with your clusters EC capability can be isolated. It is entirely possible to share devices between storage policies, but for EC it may make more sense to not only use separate devices but possibly even entire nodes dedicated for EC.

Which direction one chooses depends on why the EC policy is being deployed. If, for example, there is a production replication policy in place already and the goal is to add a cold storage tier such that the existing nodes performing replication are impacted as little as possible, adding a new set of nodes dedicated to EC might make the most sense but also incurs the most cost. On the other hand, if EC is being added as a capability to provide additional durability for a specific set of applications and the existing infrastructure is well suited for EC (sufficient number of nodes, zones for the EC scheme that is chosen) then leveraging the existing infrastructure such that the EC ring shares nodes with the replication ring makes the most sense. These are some of the main considerations:

- Layout of existing infrastructure.
- Cost of adding dedicated EC nodes (or just dedicated EC devices).
- Intended usage model(s).

The Swift code base does not include any of the algorithms necessary to perform the actual encoding and decoding of data; that is left to external libraries. The Storage Policies architecture is leveraged to enable EC on a per container basis the object rings are still used to determine the placement of EC data fragments. Although there are several code paths that are unique to an operation associated with an EC policy, an external dependency to an Erasure Code library is what Swift counts on to perform the low level EC functions. The use of an external library allows for maximum flexibility as there are a significant number of options out there, each with its owns pros and cons that can vary greatly from one use case to another.

PyECLib: External Erasure Code Library

PyECLib is a Python Erasure Coding Library originally designed and written as part of the effort to add EC support to the Swift project, however it is an independent project. The library provides a well-defined and simple Python interface and internally implements a plug-in architecture allowing it to take advantage of many well-known C libraries such as:

- Jerasure and GFComplete at http://jerasure.org.
- Intel(R) ISA-L at http://01.org/intel%C2%AE-storage-acceleration-library-open-source-version.
- Or write your own!

PyECLib uses a C based library called liberasurecode to implement the plug in infrastructure; liberasurecode is available at:

- liberasurecode: https://github.com/openstack/liberasurecode

PyECLib itself therefore allows for not only choice but further extensibility as well. PyECLib also comes with a handy utility to help determine the best algorithm to use based on the equipment that will be used

(processors and server configurations may vary in performance per algorithm). More on this will be covered in the configuration section. PyECLib is included as a Swift requirement.

For complete details see PyECLib

Storing and Retrieving Objects

We will discuss the details of how PUT and GET work in the Under the Hood section later on. The key point here is that all of the erasure code work goes on behind the scenes; this summary is a high level information overview only.

The PUT flow looks like this:

1. The proxy server streams in an object and buffers up a segment of data (size is configurable).
2. The proxy server calls on PyECLib to encode the data into smaller fragments.
3. The proxy streams the encoded fragments out to the storage nodes based on ring locations.
4. Repeat until the client is done sending data.
5. The client is notified of completion when a quorum is met.

The GET flow looks like this:

1. The proxy server makes simultaneous requests to participating nodes.
2. As soon as the proxy has the fragments it needs, it calls on PyECLib to decode the data.
3. The proxy streams the decoded data it has back to the client.
4. Repeat until the proxy is done sending data back to the client.

It may sound like, from this high level overview, that using EC is going to cause an explosion in the number of actual files stored in each nodes local file system. Although it is true that more files will be stored (because an object is broken into pieces), the implementation works to minimize this where possible, more details are available in the Under the Hood section.

Handoff Nodes

In EC policies, similarly to replication, handoff nodes are a set of storage nodes used to augment the list of primary nodes responsible for storing an erasure coded object. These handoff nodes are used in the event that one or more of the primaries are unavailable. Handoff nodes are still selected with an attempt to achieve maximum separation of the data being placed.

Reconstruction

For an EC policy, reconstruction is analogous to the process of replication for a replication type policy — essentially the reconstructor replaces the replicator for EC policy types. The basic framework of reconstruction is very similar to that of replication with a few notable exceptions:

- Because EC does not actually replicate partitions, it needs to operate at a finer granularity than what is provided with rsync, therefore EC leverages much of ssync behind the scenes (you do not need to manually configure ssync).

- Once a pair of nodes has determined the need to replace a missing object fragment, instead of pushing over a copy like replication would do, the reconstructor has to read in enough surviving fragments from other nodes and perform a local reconstruction before it has the correct data to push to the other node.
- A reconstructor does not talk to all other reconstructors in the set of nodes responsible for an EC partition, this would be far too chatty, instead each reconstructor is responsible for syncing with the partitions closest two neighbors (closest meaning left and right on the ring).

Note: EC work (encode and decode) takes place both on the proxy nodes, for PUT/GET operations, as well as on the storage nodes for reconstruction. As with replication, reconstruction can be the result of rebalancing, bit-rot, drive failure or reverting data from a hand-off node back to its primary.

2.16.2 Performance Considerations

In general, EC has different performance characteristics than replicated data. EC requires substantially more CPU to read and write data, and is more suited for larger objects that are not frequently accessed (e.g. backups).

Operators are encouraged to characterize the performance of various EC schemes and share their observations with the developer community.

2.16.3 Using an Erasure Code Policy

To use an EC policy, the administrator simply needs to define an EC policy in *swift.conf* and create/configure the associated object ring. An example of how an EC policy can be setup is shown below:

```
[storage-policy:2]
name = ec104
policy_type = erasure_coding
ec_type = liberasurecode_rs_vand
ec_num_data_fragments = 10
ec_num_parity_fragments = 4
ec_object_segment_size = 1048576
```

Lets take a closer look at each configuration parameter:

- `name`: This is a standard storage policy parameter. See *Storage Policies* for details.
- `policy_type`: Set this to `erasure_coding` to indicate that this is an EC policy.
- `ec_type`: Set this value according to the available options in the selected PyECLib back-end. This specifies the EC scheme that is to be used. For example the option shown here selects Vandermonde Reed-Solomon encoding while an option of `flat_xor_hd_3` would select Flat-XOR based HD combination codes. See the PyECLib page for full details.
- `ec_num_data_fragments`: The total number of fragments that will be comprised of data.
- `ec_num_parity_fragments`: The total number of fragments that will be comprised of parity.
- `ec_object_segment_size`: The amount of data that will be buffered up before feeding a segment into the encoder/decoder. The default value is 1048576.

When PyECLib encodes an object, it will break it into N fragments. However, what is important during configuration, is how many of those are data and how many are parity. So in the example above, PyECLib will actually break an object in 14 different fragments, 10 of them will be made up of actual object data and 4 of them will be made of parity data (calculations depending on ec_type).

When deciding which devices to use in the EC policys object ring, be sure to carefully consider the performance impacts. Running some performance benchmarking in a test environment for your configuration is highly recommended before deployment.

To create the EC policys object ring, the only difference in the usage of the `swift-ring-builder create` command is the `replicas` parameter. The `replicas` value is the number of fragments spread across the object servers associated with the ring; `replicas` must be equal to the sum of `ec_num_data_fragments` and `ec_num_parity_fragments`. For example:

```
swift-ring-builder object-1.builder create 10 14 1
```

Note that in this example the `replicas` value of `14` is based on the sum of `10` EC data fragments and `4` EC parity fragments.

Once you have configured your EC policy in *swift.conf* and created your object ring, your application is ready to start using EC simply by creating a container with the specified policy name and interacting as usual.

Note: Its important to note that once you have deployed a policy and have created objects with that policy, these configurations options cannot be changed. In case a change in the configuration is desired, you must create a new policy and migrate the data to a new container.

Warning: Using `isa_l_rs_vand` with more than 4 parity fragments creates fragments which may in some circumstances fail to reconstruct properly or (with liberasurecode < 1.3.1) reconstruct corrupted data. New policies that need large numbers of parity fragments should consider using `isa_l_rs_cauchy`. Any existing affected policies must be marked deprecated, and data in containers with that policy should be migrated to a new policy.

Migrating Between Policies

A common usage of EC is to migrate less commonly accessed data from a more expensive but lower latency policy such as replication. When an application determines that it wants to move data from a replication policy to an EC policy, it simply needs to move the data from the replicated container to an EC container that was created with the target durability policy.

2.16.4 Global EC

The following recommendations are made when deploying an EC policy that spans multiple regions in a *Global Cluster*:

- The global EC policy should use *EC Duplication* in conjunction with a *Composite Ring*, as described below.
- Proxy servers should be *configured to use read affinity* to prefer reading from their local region for the global EC policy. *Per policy configuration* allows this to be configured for individual policies.

Note: Before deploying a Global EC policy, consideration should be given to the *Known Issues*, in particular the relatively poor performance anticipated from the object-reconstructor.

EC Duplication

EC Duplication enables Swift to make duplicated copies of fragments of erasure coded objects. If an EC storage policy is configured with a non-default `ec_duplication_factor` of `N > 1`, then the policy will create `N` duplicates of each unique fragment that is returned from the configured EC engine.

Duplication of EC fragments is optimal for Global EC storage policies, which require dispersion of fragment data across failure domains. Without fragment duplication, common EC parameters will not distribute enough unique fragments between large failure domains to allow for a rebuild using fragments from any one domain. For example a uniformly distributed `10+4` EC policy schema would place 7 fragments in each of two failure domains, which is less in each failure domain than the 10 fragments needed to rebuild a missing fragment.

Without fragment duplication, an EC policy schema must be adjusted to include additional parity fragments in order to guarantee the number of fragments in each failure domain is greater than the number required to rebuild. For example, a uniformly distributed `10+18` EC policy schema would place 14 fragments in each of two failure domains, which is more than sufficient in each failure domain to rebuild a missing fragment. However, empirical testing has shown encoding a schema with `num_parity > num_data` (such as `10+18`) is less efficient than using duplication of fragments. EC fragment duplication enables Swifts Global EC to maintain more independence between failure domains without sacrificing efficiency on read/write or rebuild!

The `ec_duplication_factor` option may be configured in *swift.conf* in each `storage-policy` section. The option may be omitted - the default value is 1 (i.e. no duplication):

```
[storage-policy:2]
name = ec104
policy_type = erasure_coding
ec_type = liberasurecode_rs_vand
ec_num_data_fragments = 10
ec_num_parity_fragments = 4
ec_object_segment_size = 1048576
ec_duplication_factor = 2
```

Warning: EC duplication is intended for use with Global EC policies. To ensure independent availability of data in all regions, the `ec_duplication_factor` option should only be used in conjunction with *Composite Rings*, as described in this document.

In this example, a `10+4` schema and a duplication factor of `2` will result in `(10+4)x2 = 28` fragments being stored (we will use the shorthand `10+4x2` to denote that policy configuration) . The ring for this policy should be configured with 28 replicas (i.e. `(ec_num_data_fragments + ec_num_parity_fragments) * ec_duplication_factor`). A `10+4x2` schema **can** allow a multi-region deployment to rebuild an object to full durability even when *more* than 14 fragments are unavailable. This is advantageous with respect to a `10+18` configuration not only because reads from data fragments will be more common and more efficient, but also because a `10+4x2` can grow into a `10+4x3` to expand into another region.

EC duplication with composite rings

It is recommended that EC Duplication is used with *Composite Rings* in order to disperse duplicate fragments across regions.

When EC duplication is used, it is highly desirable to have one duplicate of each fragment placed in each region. This ensures that a set of `ec_num_data_fragments` unique fragments (the minimum needed to reconstruct an object) can always be assembled from a single region. This in turn means that objects are robust in the event of an entire region becoming unavailable.

This can be achieved by using a *composite ring* with the following properties:

- The number of component rings in the composite ring is equal to the `ec_duplication_factor` for the policy.
- Each *component* ring has a number of `replicas` that is equal to the sum of `ec_num_data_fragments` and `ec_num_parity_fragments`.
- Each component ring is populated with devices in a unique region.

This arrangement results in each component ring in the composite ring, and therefore each region, having one copy of each fragment.

For example, consider a Swift cluster with two regions, `region1` and `region2` and a `4+2x2` EC policy schema. This policy should use a composite ring with two component rings, `ring1` and `ring2`, having devices exclusively in regions `region1` and `region2` respectively. Each component ring should have `replicas = 6`. As a result, the first 6 fragments for an object will always be placed in `ring1` (i.e. in `region1`) and the second 6 duplicate fragments will always be placed in `ring2` (i.e. in `region2`).

Conversely, a conventional ring spanning the two regions may give a suboptimal distribution of duplicates across the regions; it is possible for duplicates of the same fragment to be placed in the same region, and consequently for another region to have no copies of that fragment. This may make it impossible to assemble a set of `ec_num_data_fragments` unique fragments from a single region. For example, the conventional ring could have a pathologically sub-optimal placement such as:

```
r1
  <timestamp>#0#d.data
  <timestamp>#0#d.data
  <timestamp>#2#d.data
```

(continues on next page)

(continued from previous page)

```
  <timestamp>#2#d.data
  <timestamp>#4#d.data
  <timestamp>#4#d.data
r2
  <timestamp>#1#d.data
  <timestamp>#1#d.data
  <timestamp>#3#d.data
  <timestamp>#3#d.data
  <timestamp>#5#d.data
  <timestamp>#5#d.data
```

In this case, the object cannot be reconstructed from a single region; `region1` has only the fragments with index `0, 2, 4` and `region2` has the other 3 indexes, but we need 4 unique indexes to be able to rebuild an object.

Node Selection Strategy for Reads

Proxy servers require a set of *unique* fragment indexes to decode the original object when handling a GET request to an EC policy. With a conventional EC policy, this is very likely to be the outcome of reading fragments from a random selection of backend nodes. With an EC Duplication policy it is significantly more likely that responses from a *random* selection of backend nodes might include some duplicated fragments.

For this reason it is strongly recommended that EC Duplication always be deployed in combination with *Composite Rings* and *proxy server read affinity*.

Under normal conditions with the recommended deployment, read affinity will cause a proxy server to first attempt to read fragments from nodes in its local region. These fragments are guaranteed to be unique with respect to each other. Even if there are a small number of local failures, unique local parity fragments will make up the difference. However, should enough local primary storage nodes fail, such that sufficient unique fragments are not available in the local region, a global EC cluster will proceed to read fragments from the other region(s). Random reads from the remote region are not guaranteed to return unique fragments; with EC Duplication there is a significantly high probability that the proxy server will encounter a fragment that is a duplicate of one it has already found in the local region. The proxy server will ignore these and make additional requests until it accumulates the required set of unique fragments, potentially searching all the primary and handoff locations in the local and remote regions before ultimately failing the read.

A global EC deployment configured as recommended is therefore extremely resilient. However, under extreme failure conditions read handling can be inefficient because nodes in other regions are guaranteed to have some fragments which are duplicates of those the proxy server has already received. Work is in progress to improve the proxy server node selection strategy such that when it is necessary to read from other regions, nodes that are likely to have useful fragments are preferred over those that are likely to return a duplicate.

Known Issues

Efficient Cross Region Rebuild

Work is also in progress to improve the object-reconstructor efficiency for Global EC policies. Unlike the proxy server, the reconstructor does not apply any read affinity settings when gathering fragments. It is therefore likely to receive duplicated fragments (i.e. make wasted backend GET requests) while performing *every* fragment reconstruction.

Additionally, other reconstructor optimisations for Global EC are under investigation:

- Since fragments are duplicated between regions it may in some cases be more attractive to restore failed fragments from their duplicates in another region instead of rebuilding them from other fragments in the local region.
- Conversely, to avoid WAN transfer it may be more attractive to rebuild fragments from local parity.
- During rebalance it will always be more attractive to revert a fragment from its old-primary to its new primary rather than rebuilding or transferring a duplicate from the remote region.

2.16.5 Under the Hood

Now that weve explained a little about EC support in Swift and how to configure and use it, lets explore how EC fits in at the nuts-n-bolts level.

Terminology

The term fragment has been used already to describe the output of the EC process (a series of fragments) however we need to define some other key terms here before going any deeper. Without paying special attention to using the correct terms consistently, it is very easy to get confused in a hurry!

- **chunk**: HTTP chunks received over wire (term not used to describe any EC specific operation).
- **segment**: Not to be confused with SLO/DLO use of the word, in EC we call a segment a series of consecutive HTTP chunks buffered up before performing an EC operation.
- **fragment**: Data and parity fragments are generated when erasure coding transformation is applied to a segment.
- **EC archive**: A concatenation of EC fragments; to a storage node this looks like an object.
- **ec_ndata**: Number of EC data fragments.
- **ec_nparity**: Number of EC parity fragments.

Middleware

Middleware remains unchanged. For most middleware (e.g., SLO/DLO) the fact that the proxy is fragmenting incoming objects is transparent. For list endpoints, however, it is a bit different. A caller of list endpoints will get back the locations of all of the fragments. The caller will be unable to re-assemble the original object with this information, however the node locations may still prove to be useful information for some applications.

On Disk Storage

EC archives are stored on disk in their respective objects-N directory based on their policy index. See *Storage Policies* for details on per policy directory information.

In addition to the object timestamp, the filenames of EC archives encode other information related to the archive:

- The fragment archive index. This is required for a few reasons. For one, it allows us to store fragment archives of different indexes on the same storage node which is not typical however it is possible in many circumstances. Without unique filenames for the different EC archive files in a set, we would be at risk of overwriting one archive of index *n* with another of index *m* in some scenarios.

  The index is appended to the filename just before the `.data` extension. For example, the filename for a fragment archive storing the 5th fragment would be:

  ```
  1418673556.92690#5.data
  ```

- The durable state of the archive. The meaning of this will be described in more detail later, but a fragment archive that is considered durable has an additional `#d` string included in its filename immediately before the `.data` extension. For example:

  ```
  1418673556.92690#5#d.data
  ```

A policy-specific transformation function is therefore used to build the archive filename. These functions are implemented in the diskfile module as methods of policy specific sub classes of `BaseDiskFileManager`.

The transformation function for the replication policy is simply a NOP.

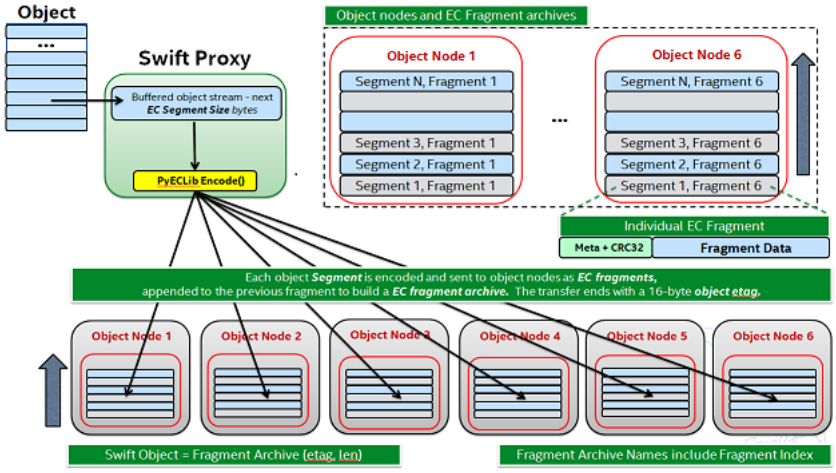
Note: In older versions the durable state of an archive was represented by an additional file called the `.durable` file instead of the `#d` substring in the `.data` filename. The `.durable` for the example above would be:

```
1418673556.92690.durable
```

Proxy Server

High Level

The Proxy Server handles Erasure Coding in a different manner than replication, therefore there are several code paths unique to EC policies either though sub classing or simple conditionals. Taking a closer look at the PUT and the GET paths will help make this clearer. But first, a high level overview of how an object flows through the system:



Note how:

- Incoming objects are buffered into segments at the proxy.
- Segments are erasure coded into fragments at the proxy.
- The proxy stripes fragments across participating nodes such that the on-disk stored files that we call a fragment archive is appended with each new fragment.

This scheme makes it possible to minimize the number of on-disk files given our segmenting and fragmenting.

Multi_Phase Conversation

Multi-part MIME document support is used to allow the proxy to engage in a handshake conversation with the storage node for processing PUT requests. This is required for a few different reasons.

1. From the perspective of the storage node, a fragment archive is really just another object, we need a mechanism to send down the original object etag after all fragment archives have landed.
2. Without introducing strong consistency semantics, the proxy needs a mechanism to know when a quorum of fragment archives have actually made it to disk before it can inform the client of a successful PUT.

MIME supports a conversation between the proxy and the storage nodes for every PUT. This provides us with the ability to handle a PUT in one connection and assure that we have the essence of a 2 phase commit, basically having the proxy communicate back to the storage nodes once it has confirmation that a quorum of fragment archives in the set have been written.

For the first phase of the conversation the proxy requires a quorum of *ec_ndata + 1* fragment archives to be successfully put to storage nodes. This ensures that the object could still be reconstructed even if one of the fragment archives becomes unavailable. As described above, each fragment archive file is named:

```
<ts>#<frag_index>.data
```

where `ts` is the timestamp and `frag_index` is the fragment archive index.

During the second phase of the conversation the proxy communicates a confirmation to storage nodes that the fragment archive quorum has been achieved. This causes each storage node to rename the fragment archive written in the first phase of the conversation to include the substring `#d` in its name:

```
<ts>#<frag_index>#d.data
```

This indicates to the object server that this fragment archive is *durable* and that there is a set of data files that are durable at timestamp `ts`.

For the second phase of the conversation the proxy requires a quorum of *ec_ndata + 1* successful commits on storage nodes. This ensures that there are sufficient committed fragment archives for the object to be reconstructed even if one becomes unavailable. The reconstructor ensures that the durable state is replicated on storage nodes where it may be missing.

Note that the completion of the commit phase of the conversation is also a signal for the object server to go ahead and immediately delete older timestamp files for this object. This is critical as we do not want to delete the older object until the storage node has confirmation from the proxy, via the multi-phase conversation, that the other nodes have landed enough for a quorum.

The basic flow looks like this:

1. The Proxy Server erasure codes and streams the object fragments (ec_ndata + ec_nparity) to the storage nodes.
2. The storage nodes store objects as EC archives and upon finishing object data/metadata write, send a 1st-phase response to proxy.
3. Upon quorum of storage nodes responses, the proxy initiates 2nd-phase by sending commit confirmations to object servers.
4. Upon receipt of commit message, object servers rename `.data` files to include the `#d` substring, indicating successful PUT, and send a final response to the proxy server.
5. The proxy waits for *ec_ndata + 1* object servers to respond with a success (2xx) status before responding to the client with a successful status.

Here is a high level example of what the conversation looks like:

```
proxy: PUT /p/a/c/o
     Transfer-Encoding': 'chunked'
     Expect': '100-continue'
     X-Backend-Obj-Multiphase-Commit: yes
obj:   100 Continue
     X-Obj-Multiphase-Commit: yes
proxy: --MIMEboundary
     X-Document: object body
     <obj_data>
     --MIMEboundary
```

(continues on next page)

(continued from previous page)

```
     X-Document: object metadata
     Content-MD5: <footer_meta_cksum>
     <footer_meta>
     --MIMEboundary
<object server writes data, metadata to <ts>#<frag_index>.data file>
obj:   100 Continue
<quorum>
proxy: X-Document: put commit
     commit_confirmation
     --MIMEboundary--
<object server renames <ts>#<frag_index>.data to <ts>#<frag_index>#d.data>
obj:   20x
<proxy waits to receive >=2 2xx responses>
proxy: 2xx -> client
```

A few key points on the durable state of a fragment archive:

- A durable fragment archive means that there exist sufficient other fragment archives elsewhere in the cluster (durable and/or non-durable) to reconstruct the object.
- When a proxy does a GET, it will require at least one object server to respond with a fragment archive is durable before reconstructing and returning the object to the client.

Partial PUT Failures

A partial PUT failure has a few different modes. In one scenario the Proxy Server is alive through the entire PUT conversation. This is a very straightforward case. The client will receive a good response if and only if a quorum of fragment archives were successfully landed on their storage nodes. In this case the Reconstructor will discover the missing fragment archives, perform a reconstruction and deliver those fragment archives to their nodes.

The more interesting case is what happens if the proxy dies in the middle of a conversation. If it turns out that a quorum had been met and the commit phase of the conversation finished, its as simple as the previous case in that the reconstructor will repair things. However, if the commit didnt get a chance to happen then some number of the storage nodes have .data files on them (fragment archives) but none of them knows whether there are enough elsewhere for the entire object to be reconstructed. In this case the client will not have received a 2xx response so there is no issue there, however, it is left to the storage nodes to clean up the stale fragment archives. Work is ongoing in this area to enable the proxy to play a role in reviving these fragment archives, however, for the current release, a proxy failure after the start of a conversation but before the commit message will simply result in a PUT failure.

GET

The GET for EC is different enough from replication that subclassing the *BaseObjectController* to the *ECObjectController* enables an efficient way to implement the high level steps described earlier:

1. The proxy server makes simultaneous requests to *ec_ndata* primary object server nodes with goal of finding a set of *ec_ndata* distinct EC archives at the same timestamp, and an indication from at least one object server that a durable fragment archive exists for that timestamp. If this goal is not achieved with the first *ec_ndata* requests then the proxy server continues to issue requests to the remaining primary nodes and then handoff nodes.
2. As soon as the proxy server has found a usable set of *ec_ndata* EC archives, it starts to call PyECLib to decode fragments as they are returned by the object server nodes.
3. The proxy server creates Etag and content length headers for the client response since each EC archives metadata is valid only for that archive.
4. The proxy streams the decoded data it has back to the client.

Note that the proxy does not require all objects servers to have a durable fragment archive to return in response to a GET. The proxy will be satisfied if just one object server has a durable fragment archive at the same timestamp as EC archives returned from other object servers. This means that the proxy can successfully GET an object that had missing durable state on some nodes when it was PUT (i.e. a partial PUT failure occurred).

Note also that an object server may inform the proxy server that it has more than one EC archive for different timestamps and/or fragment indexes, which may cause the proxy server to issue multiple requests for distinct EC archives to that object server. (This situation can temporarily occur after a ring rebalance when a handoff node storing an archive has become a primary node and received its primary archive but not yet moved the handoff archive to its primary node.)

The proxy may receive EC archives having different timestamps, and may receive several EC archives having the same index. The proxy therefore ensures that it has sufficient EC archives with the same timestamp and distinct fragment indexes before considering a GET to be successful.

Object Server

The Object Server, like the Proxy Server, supports MIME conversations as described in the proxy section earlier. This includes processing of the commit message and decoding various sections of the MIME document to extract the footer which includes things like the entire object etag.

DiskFile

Erasure code policies use subclassed `ECDiskFile`, `ECDiskFileWriter`, `ECDiskFileReader` and `ECDiskFileManager` to implement EC specific handling of on disk files. This includes things like file name manipulation to include the fragment index and durable state in the filename, construction of EC specific `hashes.pkl` file to include fragment index information, etc.

Metadata

There are few different categories of metadata that are associated with EC:

System Metadata: EC has a set of object level system metadata that it attaches to each of the EC archives. The metadata is for internal use only:

- `X-Object-Sysmeta-EC-Etag`: The Etag of the original object.
- `X-Object-Sysmeta-EC-Content-Length`: The content length of the original object.
- `X-Object-Sysmeta-EC-Frag-Index`: The fragment index for the object.
- `X-Object-Sysmeta-EC-Scheme`: Description of the EC policy used to encode the object.
- `X-Object-Sysmeta-EC-Segment-Size`: The segment size used for the object.

User Metadata: User metadata is unaffected by EC, however, a full copy of the user metadata is stored with every EC archive. This is required as the reconstructor needs this information and each reconstructor only communicates with its closest neighbors on the ring.

PyECLib Metadata: PyECLib stores a small amount of metadata on a per fragment basis. This metadata is not documented here as it is opaque to Swift.

Database Updates

As account and container rings are not associated with a Storage Policy, there is no change to how these database updates occur when using an EC policy.

The Reconstructor

The Reconstructor performs analogous functions to the replicator:

1. Recovering from disk drive failure.
2. Moving data around because of a rebalance.
3. Reverting data back to a primary from a handoff.
4. Recovering fragment archives from bit rot discovered by the auditor.

However, under the hood it operates quite differently. The following are some of the key elements in understanding how the reconstructor operates.

Unlike the replicator, the work that the reconstructor does is not always as easy to break down into the 2 basic tasks of synchronize or revert (move data from handoff back to primary) because of the fact that one storage node can house fragment archives of various indexes and each index really "belongs" to a different node. So, whereas when the replicator is reverting data from a handoff it has just one node to send its data to, the reconstructor can have several. Additionally, it is not always the case that the processing of a particular suffix directory means one or the other job type for the entire directory (as it does for replication). The scenarios that create these mixed situations can be pretty complex so we will just focus on what the reconstructor does here and not a detailed explanation of why.

Job Construction and Processing

Because of the nature of the work it has to do as described above, the reconstructor builds jobs for a single job processor. The job itself contains all of the information needed for the processor to execute the job which may be a synchronization or a data reversion. There may be a mix of jobs that perform both of these operations on the same suffix directory.

Jobs are constructed on a per-partition basis and then per-fragment-index basis. That is, there will be one job for every fragment index in a partition. Performing this construction "up front" like this helps minimize the interaction between nodes collecting hashes.pkl information.

Once a set of jobs for a partition has been constructed, those jobs are sent off to threads for execution. The single job processor then performs the necessary actions, working closely with ssync to carry out its instructions. For data reversion, the actual objects themselves are cleaned up via the ssync module and once that partitions set of jobs is complete, the reconstructor will attempt to remove the relevant directory structures.

Job construction must account for a variety of scenarios, including:

1. A partition directory with all fragment indexes matching the local node index. This is the case where everything is where it belongs and we just need to compare hashes and sync if needed. Here we simply sync with our partners.
2. A partition directory with at least one local fragment index and mix of others. Here we need to sync with our partners where fragment indexes matches the local_id, all others are syncd with their home nodes and then deleted.
3. A partition directory with no local fragment index and just one or more of others. Here we sync with just the home nodes for the fragment indexes that we have and then all the local archives are deleted. This is the basic handoff reversion case.

Note: A "home node" is the node where the fragment index encoded in the fragment archives filename matches the node index of a node in the primary partition list.

Node Communication

The replicators talk to all nodes who have a copy of their object, typically just 2 other nodes. For EC, having each reconstructor node talk to all nodes would incur a large amount of overhead as there will typically be a much larger number of nodes participating in the EC scheme. Therefore, the reconstructor is built to talk to its adjacent nodes on the ring only. These nodes are typically referred to as partners.

Reconstruction

Reconstruction can be thought of sort of like replication but with an extra step in the middle. The reconstructor is hard-wired to use ssync to determine what is missing and desired by the other side. However, before an object is sent over the wire it needs to be reconstructed from the remaining fragments as the local fragment is just that - a different fragment index than what the other end is asking for.

Thus, there are hooks in ssync for EC based policies. One case would be for basic reconstruction which, at a high level, looks like this:

- Determine which nodes need to be contacted to collect other EC archives needed to perform reconstruction.
- Update the etag and fragment index metadata elements of the newly constructed fragment archive.
- Establish a connection to the target nodes and give ssync a DiskFileLike class from which it can stream data.

The reader in this class gathers fragments from the nodes and uses PyECLib to reconstruct each segment before yielding data back to ssync. Essentially what this means is that data is buffered, in memory, on a per segment basis at the node performing reconstruction and each segment is dynamically reconstructed and delivered to `ssync_sender` where the `send_put()` method will ship them on over. The sender is then responsible for deleting the objects as they are sent in the case of data reversion.

The Auditor

Because the auditor already operates on a per storage policy basis, there are no specific auditor changes associated with EC. Each EC archive looks like, and is treated like, a regular object from the perspective of the auditor. Therefore, if the auditor finds bit-rot in an EC archive, it simply quarantines it and the reconstructor will take care of the rest just as the replicator does for replication policies.

2.17 Object Encryption

Swift supports the optional encryption of object data at rest on storage nodes. The encryption of object data is intended to mitigate the risk of users data being read if an unauthorised party were to gain physical access to a disk.

Note: Swifts data-at-rest encryption accepts plaintext object data from the client, encrypts it in the cluster, and stores the encrypted data. This protects object data from inadvertently being exposed if a data drive leaves the Swift cluster. If a user wishes to ensure that the plaintext data is always encrypted while in transit and in storage, it is strongly recommended that the data be encrypted before sending it to the Swift cluster. Encrypting on the client side is the only way to ensure that the data is fully encrypted for its entire lifecycle.

Encryption of data at rest is implemented by middleware that may be included in the proxy server WSGI pipeline. The feature is internal to a Swift cluster and not exposed through the API. Clients are unaware that data is encrypted by this feature internally to the Swift service; internally encrypted data should never be returned to clients via the Swift API.

The following data are encrypted while at rest in Swift:

- Object content i.e. the content of an object PUT requests body
- The entity tag (ETag) of objects that have non-zero content
- All custom user object metadata values i.e. metadata sent using X-Object-Meta- prefixed headers with PUT or POST requests

Any data or metadata not included in the list above are not encrypted, including:

- Account, container and object names
- Account and container custom user metadata values

- All custom user metadata names
- Object Content-Type values
- Object size
- System metadata

Note: This feature is intended to provide *confidentiality* of data that is at rest i.e. to protect user data from being read by an attacker that gains access to disks on which object data is stored.

This feature is not intended to prevent undetectable *modification* of user data at rest.

This feature is not intended to protect against an attacker that gains access to Swifts internal network connections, or gains access to key material or is able to modify the Swift code running on Swift nodes.

2.17.1 Deployment and operation

Encryption is deployed by adding two middleware filters to the proxy server WSGI pipeline and including their respective filter configuration sections in the *proxy-server.conf* file. *Additional steps* are required if the container sync feature is being used.

The *keymaster* and *encryption* middleware filters must be to the right of all other middleware in the pipeline apart from the final proxy-logging middleware, and in the order shown in this example:

```
<other middleware> keymaster encryption proxy-logging proxy-server

[filter:keymaster]
use = egg:swift#keymaster
encryption_root_secret = your_secret

[filter:encryption]
use = egg:swift#encryption
# disable_encryption = False
```

See the *proxy-server.conf-sample* file for further details on the middleware configuration options.

Keymaster middleware

The *keymaster* middleware must be configured with a root secret before it is used. By default the *keymaster* middleware will use the root secret configured using the `encryption_root_secret` option in the middleware filter section of the *proxy-server.conf* file, for example:

```
[filter:keymaster]
use = egg:swift#keymaster
encryption_root_secret = your_secret
```

Root secret values MUST be at least 44 valid base-64 characters and should be consistent across all proxy servers. The minimum length of 44 has been chosen because it is the length of a base-64 encoded 32 byte value.

Note: The `encryption_root_secret` option holds the master secret key used for encryption. The security of all encrypted data critically depends on this key and it should therefore be set to a high-entropy value. For example, a suitable `encryption_root_secret` may be obtained by base-64 encoding a 32 byte (or longer) value generated by a cryptographically secure random number generator.

The `encryption_root_secret` value is necessary to recover any encrypted data from the storage system, and therefore, it must be guarded against accidental loss. Its value (and consequently, the proxy-server.conf file) should not be stored on any disk that is in any account, container or object ring.

The `encryption_root_secret` value should not be changed once deployed. Doing so would prevent Swift from properly decrypting data that was encrypted using the former value, and would therefore result in the loss of that data.

One method for generating a suitable value for `encryption_root_secret` is to use the `openssl` command line tool:

```
openssl rand -base64 32
```

Separate keymaster configuration file

The `encryption_root_secret` option may alternatively be specified in a separate config file at a path specified by the `keymaster_config_path` option, for example:

```
[filter:keymaster]
use = egg:swift#keymaster
keymaster_config_path = /etc/swift/keymaster.conf
```

This has the advantage of allowing multiple processes which need to be encryption-aware (for example, proxy-server and container-sync) to share the same config file, ensuring that consistent encryption keys are used by those processes. It also allows the keymaster configuration file to have different permissions than the *proxy-server.conf* file.

A separate keymaster config file should have a `[keymaster]` section containing the `encryption_root_secret` option:

```
[keymaster]
encryption_root_secret = your_secret
```

Note: Alternative keymaster middleware is available to retrieve encryption root secrets from an *external key management system* such as Barbican rather than storing root secrets in configuration files.

Once deployed, the encryption filter will by default encrypt object data and metadata when handling PUT and POST requests and decrypt object data and metadata when handling GET and HEAD requests. COPY requests are transformed into GET and PUT requests by the *Server Side Copy* middleware before reaching the encryption middleware and as a result object data and metadata is decrypted and re-encrypted when copied.

Changing the encryption root secret

From time to time it may be desirable to change the root secret that is used to derive encryption keys for new data written to the cluster. The *keymaster* middleware allows alternative root secrets to be specified in its configuration using options of the form:

```
encryption_root_secret_<secret_id> = <secret value>
```

where `secret_id` is a unique identifier for the root secret and `secret value` is a value that meets the requirements for a root secret described above.

Only one root secret is used to encrypt new data at any moment in time. This root secret is specified using the `active_root_secret_id` option. If specified, the value of this option should be one of the configured root secret `secret_id` values; otherwise the value of `encryption_root_secret` will be taken as the default active root secret.

Note: The active root secret is only used to derive keys for new data written to the cluster. Changing the active root secret does not cause any existing data to be re-encrypted.

Existing encrypted data will be decrypted using the root secret that was active when that data was written. All previous active root secrets must therefore remain in the middleware configuration in order for decryption of existing data to succeed. Existing encrypted data will reference previous root secret by the `secret_id` so it must be kept consistent in the configuration.

Note: Do not remove or change any previously active <`secret value`> or <`secret_id`>.

For example, the following keymaster configuration file specifies three root secrets, with the value of `encryption_root_secret_2` being the current active root secret:

```
[keymaster]
active_root_secret_id = 2
encryption_root_secret = your_secret
encryption_root_secret_1 = your_secret_1
encryption_root_secret_2 = your_secret_2
```

Note: To ensure there is no loss of data availability, deploying a new key to your cluster requires a two-stage config change. First, add the new key to the `encryption_root_secret_`<`secret_id`> option and restart the proxy-server. Do this for all proxies. Next, set the `active_root_secret_id` option to the new secret id and restart the proxy. Again, do this for all proxies. This process ensures that all proxies will have the new key available for *decryption* before any proxy uses it for *encryption*.

Encryption middleware

Once deployed, the encryption filter will by default encrypt object data and metadata when handling PUT and POST requests and decrypt object data and metadata when handling GET and HEAD requests. COPY requests are transformed into GET and PUT requests by the *Server Side Copy* middleware before reaching the encryption middleware and as a result object data and metadata is decrypted and re-encrypted when copied.

Encryption Root Secret in External Key Management System

The benefits of using a dedicated system for storing the encryption root secret include the auditing and access control infrastructure that are already in place in such a system, and the fact that an encryption root secret stored in a key management system (KMS) may be backed by a hardware security module (HSM) for additional security. Another significant benefit of storing the root encryption secret in an external KMS is that it is in this case never stored on a disk in the Swift cluster.

Swift supports fetching encryption root secrets from a Barbican service or a KMIP service using the `kms_keymaster` or `kmip_keymaster` middleware respectively.

Encryption Root Secret in a Barbican KMS

Make sure the required dependencies are installed for retrieving an encryption root secret from an external KMS. This can be done when installing Swift (add the `-e` flag to install as a development version) by changing to the Swift directory and running the following command to install Swift together with the `kms_keymaster` extra dependencies:

```
sudo pip install .[kms_keymaster]
```

Another way to install the dependencies is by making sure the following lines exist in the requirements.txt file, and installing them using `pip install -r requirements.txt`:

```
cryptography>=1.6                       # BSD/Apache-2.0
castellan>=0.6.0
```

Note: If any of the required packages is already installed, the `--upgrade` flag may be required for the `pip` commands in order for the required minimum version to be installed.

To make use of an encryption root secret stored in an external KMS, replace the keymaster middleware with the kms_keymaster middleware in the proxy server WSGI pipeline in *proxy-server.conf*, in the order shown in this example:

```
<other middleware> kms_keymaster encryption proxy-logging proxy-server
```

and add a section to the same file:

```
[filter:kms_keymaster]
use = egg:swift#kms_keymaster
keymaster_config_path = file_with_kms_keymaster_config
```

Create or edit the file *file_with_kms_keymaster_config* referenced above. For further details on the middleware configuration options, see the *keymaster.conf-sample* file. An example of the content of this file, with optional parameters omitted, is below:

```
[kms_keymaster]
key_id = changeme
username = swift
password = password
project_name = swift
auth_endpoint = http://keystonehost:5000/v3
```

The encryption root secret shall be created and stored in the external key management system before it can be used by the keymaster. It shall be stored as a symmetric key, with content type `application/octet-stream`, `base64` content encoding, `AES` algorithm, bit length `256`, and secret type `symmetric`. The mode `ctr` may also be stored for informational purposes - it is not currently checked by the keymaster.

The following command can be used to store the currently configured `encryption_root_secret` value from the *proxy-server.conf* file in Barbican:

```
openstack secret store --name swift_root_secret \
--payload-content-type="application/octet-stream" \
--payload-content-encoding="base64" --algorithm aes --bit-length 256 \
--mode ctr --secret-type symmetric --payload <base64_encoded_root_secret>
```

Alternatively, the existing root secret can also be stored in Barbican using curl.

Note: The credentials used to store the secret in Barbican shall be the same ones that the proxy server uses to retrieve the secret, i.e., the ones configured in the *keymaster.conf* file. For clarity reasons the commands shown here omit the credentials - they may be specified explicitly, or in environment variables.

Instead of using an existing root secret, Barbican can also be asked to generate a new 256-bit root secret, with content type `application/octet-stream` and algorithm `AES` (the `mode` parameter is currently optional):

```
openstack secret order create --name swift_root_secret \
--payload-content-type="application/octet-stream" --algorithm aes \
--bit-length 256 --mode ctr key
```

The `order create` creates an asynchronous request to create the actual secret. The order can be retrieved using `openstack secret order get`, and once the order completes successfully, the output will show the key id of the generated root secret. Keys currently stored in Barbican can be listed using the `openstack secret list` command.

Note: Both the order (the asynchronous request for creating or storing a secret), and the actual secret itself, have similar unique identifiers. Once the order has been completed, the key id is shown in the output of the `order get` command.

The keymaster uses the explicitly configured username and password (and project name etc.) from the *keymaster.conf* file for retrieving the encryption root secret from an external key management system. The Castellan library is used to communicate with Barbican.

For the proxy server, reading the encryption root secret directly from the *proxy-server.conf* file, from the *keymaster.conf* file pointed to from the *proxy-server.conf* file, or from an external key management system such as Barbican, are all functionally equivalent. In case reading the encryption root secret from the external key management system fails, the proxy server will not start up. If the encryption root secret is retrieved successfully, it is cached in memory in the proxy server.

For further details on the configuration options, see the *[filter:kms_keymaster]* section in the *proxy-server.conf-sample* file, and the *keymaster.conf-sample* file.

Encryption Root Secret in a KMIP service

This middleware enables Swift to fetch a root secret from a KMIP service. The root secret is expected to have been previously created in the KMIP service and is referenced by its unique identifier. The secret should be an AES-256 symmetric key.

To use this middleware Swift must be installed with the extra required dependencies:

```
sudo pip install .[kmip_keymaster]
```

Add the `-e` flag to install as a development version.

Edit the swift *proxy-server.conf* file to insert the middleware in the wsgi pipeline, replacing any other keymaster middleware:

```
[pipeline:main]
pipeline = catch_errors gatekeeper healthcheck proxy-logging \
    <other middleware> kmip_keymaster encryption proxy-logging proxy-server
```

and add a new filter section:

```
[filter:kmip_keymaster]
use = egg:swift#kmip_keymaster
key_id = <unique id of secret to be fetched from the KMIP service>
host = <KMIP server host>
port = <KMIP server port>
certfile = /path/to/client/cert.pem
keyfile = /path/to/client/key.pem
ca_certs = /path/to/server/cert.pem
username = <KMIP username>
password = <KMIP password>
```

Apart from `use` and `key_id` the options are as defined for a PyKMIP client. The authoritative definition of these options can be found at https://pykmip.readthedocs.io/en/latest/client.html.

The value of the `key_id` option should be the unique identifier for a secret that will be retrieved from the KMIP service.

The keymaster configuration can alternatively be defined in a separate config file by using the `keymaster_config_path` option:

```
[filter:kmip_keymaster]
use = egg:swift#kmip_keymaster
keymaster_config_path = /etc/swift/kmip_keymaster.conf
```

In this case, the `filter:kmip_keymaster` section should contain no other options than `use` and `keymaster_config_path`. All other options should be defined in the separate config file in a section named `kmip_keymaster`. For example:

```
[kmip_keymaster]
key_id = 1234567890
host = 127.0.0.1
port = 5696
certfile = /etc/swift/kmip_client.crt
keyfile = /etc/swift/kmip_client.key
ca_certs = /etc/swift/kmip_server.crt
username = swift
password = swift_password
```

Changing the encryption root secret of external KMSs

Because the KMS and KMIP keymasters derive from the default KeyMaster they also have to ability to define multiple keys. The only difference is the key option names. Instead of using the form *encryption_root_secret_<secret_id>* both external KMSs use *key_id_<secret_id>*, as it is an extension of their existing configuration. For example:

```
...
key_id = 1234567890
key_id_foo = 0987654321
key_id_bar = 5432106789
active_root_secret_id = foo
...
```

Other then that, the process is the same as *Changing the encryption root secret*.

Upgrade Considerations

When upgrading an existing cluster to deploy encryption, the following sequence of steps is recommended:

1. Upgrade all object servers
2. Upgrade all proxy servers
3. Add keymaster and encryption middlewares to every proxy servers middleware pipeline with the encryption `disable_encryption` option set to `True` and the keymaster `encryption_root_secret` value set as described above.
4. If required, follow the steps for *Container sync configuration*.
5. Finally, change the encryption `disable_encryption` option to `False`

Objects that existed in the cluster prior to the keymaster and encryption middlewares being deployed are still readable with GET and HEAD requests. The content of those objects will not be encrypted unless they are written again by a PUT or COPY request. Any user metadata of those objects will not be encrypted unless it is written again by a PUT, POST or COPY request.

Disabling Encryption

Once deployed, the keymaster and encryption middlewares should not be removed from the pipeline. To do so will cause encrypted object data and/or metadata to be returned in response to GET or HEAD requests for objects that were previously encrypted.

Encryption of inbound object data may be disabled by setting the encryption `disable_encryption` option to `True`, in which case existing encrypted objects will remain encrypted but new data written with PUT, POST or COPY requests will not be encrypted. The keymaster and encryption middlewares should remain in the pipeline even when encryption of new objects is not required. The encryption middleware is needed to handle GET requests for objects that may have been previously encrypted. The keymaster is needed to provide keys for those requests.

Container sync configuration

If container sync is being used then the keymaster and encryption middlewares must be added to the container sync internal client pipeline. The following configuration steps are required:

1. Create a custom internal client configuration file for container sync (if one is not already in use) based on the sample file *internal-client.conf-sample*. For example, copy *internal-client.conf-sample* to */etc/swift/container-sync-client.conf*.
2. Modify this file to include the middlewares in the pipeline in the same way as described above for the proxy server.
3. Modify the container-sync section of all container server config files to point to this internal client config file using the `internal_client_conf_path` option. For example:

```
internal_client_conf_path = /etc/swift/container-sync-client.conf
```

Note: The `encryption_root_secret` value is necessary to recover any encrypted data from the storage system, and therefore, it must be guarded against accidental loss. Its value (and consequently, the custom internal client configuration file) should not be stored on any disk that is in any account, container or object ring.

Note: These container sync configuration steps will be necessary for container sync probe tests to pass if the encryption middlewares are included in the proxy pipeline of a test cluster.

2.17.2 Implementation

Encryption scheme

Plaintext data is encrypted to ciphertext using the AES cipher with 256-bit keys implemented by the python cryptography package. The cipher is used in counter (CTR) mode so that any byte or range of bytes in the ciphertext may be decrypted independently of any other bytes in the ciphertext. This enables very simple handling of ranged GETs.

In general an item of unencrypted data, `plaintext`, is transformed to an item of encrypted data, `ciphertext`:

```
ciphertext = E(plaintext, k, iv)
```

where `E` is the encryption function, `k` is an encryption key and `iv` is a unique initialization vector (IV) chosen for each encryption context. For example, the object body is one encryption context with a randomly chosen IV. The IV is stored as metadata of the encrypted item so that it is available for decryption:

```
plaintext = D(ciphertext, k, iv)
```

where `D` is the decryption function.

The implementation of CTR mode follows NIST SP800-38A, and the full IV passed to the encryption or decryption function serves as the initial counter block.

In general any encrypted item has accompanying crypto-metadata that describes the IV and the cipher algorithm used for the encryption:

```
crypto_metadata = {"iv": <16 byte value>,
                   "cipher": "AES_CTR_256"}
```

This crypto-metadata is stored either with the ciphertext (for user metadata and etags) or as a separate header (for object bodies).

Key management

A keymaster middleware is responsible for providing the keys required for each encryption and decryption operation. Two keys are required when handling object requests: a *container key* that is uniquely associated with the container path and an *object key* that is uniquely associated with the object path. These keys are made available to the encryption middleware via a callback function that the keymaster installs in the WSGI request environ.

The current keymaster implementation derives container and object keys from the `encryption_root_secret` in a deterministic way by constructing a SHA256 HMAC using the `encryption_root_secret` as a key and the container or object path as a message, for example:

```
object_key = HMAC(encryption_root_secret, "/a/c/o")
```

Other strategies for providing object and container keys may be employed by future implementations of alternative keymaster middleware.

During each object PUT, a random key is generated to encrypt the object body. This random key is then encrypted using the object key provided by the keymaster. This makes it safe to store the encrypted random key alongside the encrypted object data and metadata.

This process of *key wrapping* enables more efficient re-keying events when the object key may need to be replaced and consequently any data encrypted using that key must be re-encrypted. Key wrapping minimizes the amount of data encrypted using those keys to just other randomly chosen keys which can be re-wrapped efficiently without needing to re-encrypt the larger amounts of data that were encrypted using the random keys.

Note: Re-keying is not currently implemented. Key wrapping is implemented in anticipation of future re-keying operations.

Encryption middleware

The encryption middleware is composed of an *encrypter* component and a *decrypter* component.

Encrypter operation

Custom user metadata

The encrypter encrypts each item of custom user metadata using the object key provided by the keymaster and an IV that is randomly chosen for that metadata item. The encrypted values are stored as *Object Transient-Sysmeta* with associated crypto-metadata appended to the encrypted value. For example:

```
X-Object-Meta-Private1: value1
X-Object-Meta-Private2: value2
```

are transformed to:

```
X-Object-Transient-Sysmeta-Crypto-Meta-Private1:
  E(value1, object_key, header_iv_1); swift_meta={"iv": header_iv_1,
                                                  "cipher": "AES_CTR_256"}
X-Object-Transient-Sysmeta-Crypto-Meta-Private2:
  E(value2, object_key, header_iv_2); swift_meta={"iv": header_iv_2,
                                                  "cipher": "AES_CTR_256"}
```

The unencrypted custom user metadata headers are removed.

Object body

Encryption of an object body is performed using a randomly chosen body key and a randomly chosen IV:

```
body_ciphertext = E(body_plaintext, body_key, body_iv)
```

The body_key is wrapped using the object key provided by the keymaster and a randomly chosen IV:

```
wrapped_body_key = E(body_key, object_key, body_key_iv)
```

The encrypter stores the associated crypto-metadata in a system metadata header:

```
X-Object-Sysmeta-Crypto-Body-Meta:
    {"iv": body_iv,
     "cipher": "AES_CTR_256",
     "body_key": {"key": wrapped_body_key,
                  "iv": body_key_iv}}
```

Note that in this case there is an extra item of crypto-metadata which stores the wrapped body key and its IV.

Entity tag

While encrypting the object body the encrypter also calculates the ETag (md5 digest) of the plaintext body. This value is encrypted using the object key provided by the keymaster and a randomly chosen IV, and saved as an item of system metadata, with associated crypto-metadata appended to the encrypted value:

```
X-Object-Sysmeta-Crypto-Etag:
  E(md5(plaintext), object_key, etag_iv); swift_meta={"iv": etag_iv,
                                                      "cipher": "AES_CTR_256"}
```

The encrypter also forces an encrypted version of the plaintext ETag to be sent with container updates by adding an update override header to the PUT request. The associated crypto-metadata is appended to the encrypted ETag value of this update override header:

```
X-Object-Sysmeta-Container-Update-Override-Etag:
    E(md5(plaintext), container_key, override_etag_iv);
    meta={"iv": override_etag_iv, "cipher": "AES_CTR_256"}
```

The container key is used for this encryption so that the decrypter is able to decrypt the ETags in container listings when handling a container request, since object keys may not be available in that context.

Since the plaintext ETag value is only known once the encrypter has completed processing the entire object body, the `X-Object-Sysmeta-Crypto-Etag` and `X-Object-Sysmeta-Container-Update-Override-Etag` headers are sent after the encrypted object body using the proxy servers support for request footers.

Conditional Requests

In general, an object server evaluates conditional requests with `If[-None]-Match` headers by comparing values listed in an `If[-None]-Match` header against the ETag that is stored in the object metadata. This is not possible when the ETag stored in object metadata has been encrypted. The encrypter therefore calculates an HMAC using the object key and the ETag while handling object PUT requests, and stores this under the metadata key `X-Object-Sysmeta-Crypto-Etag-Mac`:

```
X-Object-Sysmeta-Crypto-Etag-Mac: HMAC(object_key, md5(plaintext))
```

Like other ETag-related metadata, this is sent after the encrypted object body using the proxy servers support for request footers.

The encrypter similarly calculates an HMAC for each ETag value included in `If[-None]-Match` headers of conditional GET or HEAD requests, and appends these to the `If[-None]-Match` header. The encrypter also sets the `X-Backend-Etag-Is-At` header to point to the previously stored `X-Object-Sysmeta-Crypto-Etag-Mac` metadata so that the object server evaluates the conditional request by comparing the HMAC values included in the `If[-None]-Match` with the value stored under `X-Object-Sysmeta-Crypto-Etag-Mac`. For example, given a conditional request with header:

```
If-Match: match_etag
```

the encrypter would transform the request headers to include:

```
If-Match: match_etag,HMAC(object_key, match_etag)
X-Backend-Etag-Is-At: X-Object-Sysmeta-Crypto-Etag-Mac
```

This enables the object server to perform an encrypted comparison to check whether the ETags match, without leaking the ETag itself or leaking information about the object body.

Decrypter operation

For each GET or HEAD request to an object, the decrypter inspects the response for encrypted items (revealed by crypto-metadata headers), and if any are discovered then it will:

1. Fetch the object and container keys from the keymaster via its callback
2. Decrypt the `X-Object-Sysmeta-Crypto-Etag` value
3. Decrypt the `X-Object-Sysmeta-Container-Update-Override-Etag` value
4. Decrypt metadata header values using the object key
5. Decrypt the wrapped body key found in `X-Object-Sysmeta-Crypto-Body-Meta`
6. Decrypt the body using the body key

For each GET request to a container that would include ETags in its response body, the decrypter will:

1. GET the response body with the container listing
2. Fetch the container key from the keymaster via its callback
3. Decrypt any encrypted ETag entries in the container listing using the container key

Impact on other Swift services and features

Encryption has no impact on *Versioned Writes* other than that any previously unencrypted objects will be encrypted as they are copied to or from the versions container. Keymaster and encryption middlewares should be placed after `versioned_writes` in the proxy server pipeline, as described in *Deployment and operation*.

Container Sync uses an internal client to GET objects that are to be syncd. This internal client must be configured to use the keymaster and encryption middlewares as described *above*.

Encryption has no impact on the *object-auditor* service. Since the ETag header saved with the object at rest is the md5 sum of the encrypted object body then the auditor will verify that encrypted data is valid.

Encryption has no impact on the *object-expirer* service. `X-Delete-At` and `X-Delete-After` headers are not encrypted.

Encryption has no impact on the *object-replicator* and *object-reconstructor* services. These services are unaware of the object or EC fragment data being encrypted.

Encryption has no impact on the *container-reconciler* service. The *container-reconciler* uses an internal client to move objects between different policy rings. The reconcilers pipeline *MUST NOT* have encryption enabled. The destination object has the same URL as the source object and the object is moved without re-encryption.

Considerations for developers

Developers should be aware that keymaster and encryption middlewares rely on the path of an object remaining unchanged. The included keymaster derives keys for containers and objects based on their paths and the `encryption_root_secret`. The keymaster does not rely on object metadata to inform its generation of keys for GET and HEAD requests because when handling *Conditional Requests* it is required to provide the object key before any metadata has been read from the object.

Developers should therefore give careful consideration to any new features that would relocate object data and metadata within a Swift cluster by means that do not cause the object data and metadata to pass through the encryption middlewares in the proxy pipeline and be re-encrypted.

The crypto-metadata associated with each encrypted item does include some *key_id* metadata that is provided by the keymaster and contains the path used to derive keys. This *key_id* metadata is persisted in anticipation of future scenarios when it may be necessary to decrypt an object that has been relocated without re-encrypting, in which case the metadata could be used to derive the keys that were used for encryption. However, this alone is not sufficient to handle conditional requests and to decrypt container listings where objects have been relocated, and further work will be required to solve those issues.

2.18 Using Swift as Backing Store for Service Data

2.18.1 Background

This section provides guidance to OpenStack Service developers for how to store your users data in Swift. An example of this is that a user requests that Nova save a snapshot of a VM. Nova passes the request to Glance, Glance writes the image to a Swift container as a set of objects.

Throughout this section, the following terminology and concepts are used:

- User or end-user. This is a person making a request that will result in an OpenStack Service making a request to Swift.
- Project (also known as Tenant). This is the unit of resource ownership. While data such as snapshot images or block volume backups may be stored as a result of an end-users request, the reality is that these are project data.
- Service. This is a program or system used by end-users. Specifically, it is any program or system that is capable of receiving end-users tokens and validating the token with the Keystone Service and has a need to store data in Swift. Glance and Cinder are examples of such Services.
- Service User. This is a Keystone user that has been assigned to a Service. This allows the Service to generate and use its own tokens so that it can interact with other Services as itself.
- Service Project. This is a project (tenant) that is associated with a Service. There may be a single project shared by many Services or there may be a project dedicated to each Service. In this document, the main purpose of the Service Project is to allow the system operator to configure specific roles for each Service User.

2.18.2 Alternate Backing Store Schemes

There are three schemes described here:

- Dedicated Service Account (Single Tenant)

  Your Service has a dedicated Service Project (hence a single dedicated Swift account). Data for all users and projects are stored in this account. Your Service must have a user assigned to it (the Service User). When you have data to store on behalf of one of your users, you use the Service User credentials to get a token for the Service Project and request Swift to store the data in the Service Project.

  With this scheme, data for all users is stored in a single account. This is transparent to your users and since the credentials for the Service User are typically not shared with anyone, your users cannot access their data by making a request directly to Swift. However, since data belonging to all users is stored in one account, it presents a single point of vulnerably to accidental deletion or a leak of the service-user credentials.

- Multi Project (Multi Tenant)

  Data belonging to a project is stored in the Swift account associated with the project. Users make requests to your Service using a token scoped to a project in the normal way. You can then use this same token to store the user data in the projects Swift account.

  The effect is that data is stored in multiple projects (aka tenants). Hence this scheme has been known as the multi tenant scheme.

  With this scheme, access is controlled by Keystone. The users must have a role that allows them to perform the request to your Service. In addition, they must have a role that also allows them to store data in the Swift account. By default, the admin or swiftoperator roles are used for this purpose (specific systems may use other role names). If the user does not have the appropriate roles, when your Service attempts to access Swift, the operation will fail.

  Since you are using the users token to access the data, it follows that the user can use the same token to access Swift directly bypassing your Service. When end-users are browsing containers, they will also see your Services containers and objects and may potentially delete the data. Conversely, there is no single account where all data so leakage of credentials will only affect a single project/tenant.

- Service Prefix Account

  Data belonging to a project is stored in a Swift account associated with the project. This is similar to the Multi Project scheme described above. However, the Swift account is different than the account that users access. Specifically, it has a different account prefix. For example, for the project 1234, the user account is named AUTH_1234. Your Service uses a different account, for example, SERVICE_1234.

  To access the SERVICE_1234 account, you must present two tokens: the users token is put in the X-Auth-Token header. You present your Services token in the X-Service-Token header. Swift is configured such that only when both tokens are presented will it allow access. Specifically, the user cannot bypass your Service because they only have their own token. Conversely, your Service can only access the data while it has a copy of the users token the Services token by itself will not grant access.

  The data stored in the Service Prefix Account cannot be seen by end-users. So they cannot delete this data they can only access the data if they make a request through your Service. The data is also more secure. To make an unauthorized access, someone would need to compromise both an

end-users and your Service User credentials. Even then, this would only expose one project not other projects.

The Service Prefix Account scheme combines features of the Dedicated Service Account and Multi Project schemes. It has the private, dedicated, characteristics of the Dedicated Service Account scheme but does not present a single point of attack. Using the Service Prefix Account scheme is a little more involved than the other schemes, so the rest of this document describes it more detail.

2.18.3 Service Prefix Account Overview

The following diagram shows the flow through the system from the end-user, to your Service and then onto Swift:

The sequence of events and actions are as follows:

- Request arrives at your Service
- The <user-token> is validated by the keystonemiddleware.auth_token middleware. The users role(s) are used to determine if the user can perform the request. See *The Auth System* for technical information on the authentication system.
- As part of this request, your Service needs to access Swift (either to write or read a container or object). In this example, you want to perform a PUT on <container>/<object>.
- In the wsgi environment, the auth_token module will have populated the HTTP_X_SERVICE_CATALOG item. This lists the Swift endpoint and account. This is something such as https://<netloc>/v1/AUTH_1234 where `AUTH_` is a prefix and `1234` is the project id.
- The `AUTH_` prefix is the default value. However, your system may use a different prefix. To determine the actual prefix, search for the first underscore (_) character in the account name. If there is no underscore character in the account name, this means there is no prefix.
- Your Service should have a configuration parameter that provides the appropriate prefix to use for storing data in Swift. There is more discussion of this below, but for now assume the prefix is `SERVICE_`.
- Replace the prefix (`AUTH_` in above examples) in the path with `SERVICE_`, so the full URL to access the object becomes https://<netloc>/v1/SERVICE_1234/<container>/<object>.
- Make the request to Swift, using this URL. In the X-Auth-Token header place a copy of the <user-token>. In the X-Service-Token header, place your Services token. If you use python-swiftclient

you can achieve this by:

- Putting the URL in the `preauthurl` parameter
- Putting the <user-token> in `preauthtoken` parameter
- Adding the X-Service-Token to the `headers` parameter

Using the HTTP_X_SERVICE_CATALOG to get Swift Account Name

The auth_token middleware populates the wsgi environment with information when it validates the users token. The HTTP_X_SERVICE_CATALOG item is a JSON string containing details of the OpenStack endpoints. For Swift, this also contains the projects Swift account name. Here is an example of a catalog entry for Swift:

```
"serviceCatalog": [
    ...
    {
        ....
        "type": "object-store",
        "endpoints": [
            ...
            {
                ...
                "publicURL": "https://<netloc>/v1/AUTH_1234",
                "region": "<region-name>"
                ...
            }
            ...
     ...
     }
}
```

To get the End-users account:

- Look for an entry with `type` of `object-store`
- If there are several regions, there will be several endpoints. Use the appropriate region name and select the `publicURL` item.
- The Swift account name is the final item in the path (AUTH_1234 in this example).

Getting a Service Token

A Service Token is no different than any other token and is requested from Keystone using user credentials and project in the usual way. The core requirement is that your Service User has the appropriate role. In practice:

- Your Service must have a user assigned to it (the Service User).
- Your Service has a project assigned to it (the Service Project).
- The Service User must have a role on the Service Project. This role is distinct from any of the normal end-user roles.

- The role used must the role configured in the /etc/swift/proxy-server.conf. This is the <prefix>_service_roles option. In this example, the role is the service role:

```
[keystoneauth]
reseller_prefix = AUTH_, SERVICE_
SERVICE_service_role = service
```

The service role should only be granted to OpenStack Services. It should not be granted to users.

Single or multiple Service Prefixes?

Most of the examples used in this document used a single prefix. The prefix, SERVICE was used. By using a single prefix, an operator is allowing all OpenStack Services to share the same account for data associated with a given project. For test systems or deployments well protected on private firewalled networks, this is appropriate.

However, if one Service is compromised, that Service can access data created by another Service. To prevent this, multiple Service Prefixes may be used. This also requires that the operator configure multiple service roles. For example, in a system that has Glance and Cinder, the following Swift configuration could be used:

```
[keystoneauth]
reseller_prefix = AUTH_, IMAGE_, BLOCK_
IMAGE_service_roles = image_service
BLOCK_service_roles = block_service
```

The Service User for Glance would be granted the image_service role on its Service Project and the Cinder Service user is granted the block_service role on its project. In this scheme, if the Cinder Service was compromised, it would not be able to access any Glance data.

Container Naming

Since a single Service Prefix is possible, container names should be prefixed with a unique string to prevent name clashes. We suggest you use the service type field (as used in the service catalog). For example, The Glance Service would use image as a prefix.

2.19 Container Sharding

Container sharding is an operator controlled feature that may be used to shard very large container databases into a number of smaller shard containers

---

Note: It is strongly recommended that operators gain experience of sharding containers in a non-production cluster before using in production.

The sharding process involves moving all sharding container database records via the container replication engine; the time taken to complete sharding is dependent upon the existing cluster load and the performance of the container database being sharded.

There is currently no documented process for reversing the sharding process once sharding has been enabled.

2.19.1 Background

The metadata for each container in Swift is stored in an SQLite database. This metadata includes: information about the container such as its name, modification time and current object count; user metadata that may been written to the container by clients; a record of every object in the container. The container database object records are used to generate container listings in response to container GET requests; each object record stores the objects name, size, hash and content-type as well as associated timestamps.

As the number of objects in a container increases then the number of object records in the container database increases. Eventually the container database performance starts to degrade and the time taken to update an object record increases. This can result in object updates timing out, with a corresponding increase in the backlog of pending *asynchronous updates* on object servers. Container databases are typically replicated on several nodes and any database performance degradation can also result in longer *container replication* times.

The point at which container database performance starts to degrade depends upon the choice of hardware in the container ring. Anecdotal evidence suggests that containers with tens of millions of object records have noticeably degraded performance.

This performance degradation can be avoided by ensuring that clients use an object naming scheme that disperses objects across a number of containers thereby distributing load across a number of container databases. However, that is not always desirable nor is it under the control of the cluster operator.

Swifts container sharding feature provides the operator with a mechanism to distribute the load on a single client-visible container across multiple, hidden, shard containers, each of which stores a subset of the containers object records. Clients are unaware of container sharding; clients continue to use the same API to access a container that, if sharded, maps to a number of shard containers within the Swift cluster.

2.19.2 Deployment and operation

Upgrade Considerations

It is essential that all servers in a Swift cluster have been upgraded to support the container sharding feature before attempting to shard a container.

Identifying containers in need of sharding

Container sharding is currently initiated by the `swift-manage-shard-ranges` CLI tool *described below*. Operators must first identify containers that are candidates for sharding. To assist with this, the *container-sharder daemon* inspects the size of containers that it visits and writes a list of sharding candidates to recon cache. For example:

```
"sharding_candidates": {
    "found": 1,
    "top": [
        {
            "account": "AUTH_test",
```

(continues on next page)

(continued from previous page)

```
            "container": "c1",
            "file_size": 497763328,
            "meta_timestamp": "1525346445.31161",
            "node_index": 2,
            "object_count": 3349028,
            "path": <path_to_db>,
            "root": "AUTH_test/c1"
        }
    ]
}
```

A container is considered to be a sharding candidate if its object count is greater than or equal to the `shard_container_threshold` option. The number of candidates reported is limited to a number configured by the `recon_candidates_limit` option such that only the largest candidate containers are included in the `sharding_candidates` data.

`swift-manage-shard-ranges` CLI tool

The `swift-manage-shard-ranges` tool provides commands for initiating sharding of a container. `swift-manage-shard-ranges` operates directly on a container database file.

Note: `swift-manage-shard-ranges` must only be used on one replica of a container database to avoid inconsistent results. The modifications made by `swift-manage-shard-ranges` will be automatically copied to other replicas of the container database via normal replication processes.

There are three steps in the process of initiating sharding, each of which may be performed in isolation or, as shown below, using a single command.

1. The `find` sub-command scans the container database to identify how many shard containers will be required and which objects they will manage. Each shard container manages a range of the object namespace defined by a `lower` and `upper` bound. The maximum number of objects to be allocated to each shard container is specified on the command line. For example:

```
$ swift-manage-shard-ranges <path_to_db> find 500000
Loaded db broker for AUTH_test/c1.
[
  {
    "index": 0,
    "lower": "",
    "object_count": 500000,
    "upper": "o_01086834"
  },
  {
    "index": 1,
    "lower": "o_01086834",
    "object_count": 500000,
    "upper": "o_01586834"
  },
```

(continues on next page)

(continued from previous page)

```
  {
    "index": 2,
    "lower": "o_01586834",
    "object_count": 500000,
    "upper": "o_02087570"
  },
  {
    "index": 3,
    "lower": "o_02087570",
    "object_count": 500000,
    "upper": "o_02587572"
  },
  {
    "index": 4,
    "lower": "o_02587572",
    "object_count": 500000,
    "upper": "o_03087572"
  },
  {
    "index": 5,
    "lower": "o_03087572",
    "object_count": 500000,
    "upper": "o_03587572"
  },
  {
    "index": 6,
    "lower": "o_03587572",
    "object_count": 349194,
    "upper": ""
  }
]
Found 7 ranges in 4.37222s (total object count 3349194)
```

This command returns a list of shard ranges each of which describes the namespace to be managed by a shard container. No other action is taken by this command and the container database is unchanged. The output may be redirected to a file for subsequent retrieval by the `replace` command. For example:

```
$ swift-manage-shard-ranges <path_to_db> find 500000 > my_shard_ranges
Loaded db broker for AUTH_test/c1.
Found 7 ranges in 2.448s (total object count 3349194)
```

2. The `replace` sub-command deletes any shard ranges that might already be in the container database and inserts shard ranges from a given file. The file contents should be in the format generated by the `find` sub-command. For example:

```
$ swift-manage-shard-ranges <path_to_db> replace my_shard_ranges
Loaded db broker for AUTH_test/c1.
No shard ranges found to delete.
Injected 7 shard ranges.
```

(continues on next page)

(continued from previous page)

```
Run container-replicator to replicate them to other nodes.
Use the enable sub-command to enable sharding.
```

The container database is modified to store the shard ranges, but the container will not start sharding until sharding is enabled. The `info` sub-command may be used to inspect the state of the container database at any point, and the `show` sub-command may be used to display the inserted shard ranges.

Shard ranges stored in the container database may be replaced using the `replace` sub-command. This will first delete all existing shard ranges before storing new shard ranges. Shard ranges may also be deleted from the container database using the `delete` sub-command.

Shard ranges should not be replaced or deleted using `swift-manage-shard-ranges` once the next step of enabling sharding has been taken.

3. The `enable` sub-command enables the container for sharding. The sharder daemon and/or container replicator daemon will replicate shard ranges to other replicas of the container DB and the sharder daemon will proceed to shard the container. This process may take some time depending on the size of the container, the number of shard ranges and the underlying hardware.

Note: Once the `enable` sub-command has been used there is no supported mechanism to revert sharding. Do not use `swift-manage-shard-ranges` to make any further changes to the shard ranges in the container DB.

For example:

```
$ swift-manage-shard-ranges <path_to_db> enable
Loaded db broker for AUTH_test/c1.
Container moved to state 'sharding' with epoch 1525345093.22908.
Run container-sharder on all nodes to shard the container.
```

This does not shard the container - sharding is performed by the *container-sharder daemon* - but sets the necessary state in the database for the daemon to subsequently start the sharding process.

The `epoch` value displayed in the output is the time at which sharding was enabled. When the *container-sharder daemon* starts sharding this container it creates a new container database file using the epoch in the filename to distinguish it from the retiring DB that is being sharded.

All three steps may be performed with one sub-command:

```
$ swift-manage-shard-ranges <path_to_db> find_and_replace 500000 --enable --
↪force
Loaded db broker for AUTH_test/c1.
No shard ranges found to delete.
Injected 7 shard ranges.
Run container-replicator to replicate them to other nodes.
Container moved to state 'sharding' with epoch 1525345669.46153.
Run container-sharder on all nodes to shard the container.
```

exception swift.cli.manage_shard_ranges.**GapsFoundException**

Bases: *swift.cli.manage_shard_ranges.ManageShardRangesException*

exception swift.cli.manage_shard_ranges.**InvalidSolutionException**(*msg*, *acceptor_path*, *overlapping_donors*)

Bases: *swift.cli.manage_shard_ranges.ManageShardRangesException*

exception swift.cli.manage_shard_ranges.**InvalidStateException**

Bases: *swift.cli.manage_shard_ranges.ManageShardRangesException*

exception swift.cli.manage_shard_ranges.**ManageShardRangesException**

Bases: Exception

container-sharder daemon

Once sharding has been enabled for a container, the act of sharding is performed by the *Container Sharder*. The *Container Sharder* daemon must be running on all container servers. The container-sharder daemon periodically visits each container database to perform any container sharding tasks that are required.

The container-sharder daemon requires a [container-sharder] config section to exist in the container server configuration file; a sample config section is shown in the *container-server.conf-sample* file.

Note: The auto_shard option is currently **NOT** recommended for production systems and should be set to false (the default value).

Several of the [container-sharder] config options are only significant when the auto_shard option is enabled. This option enables the container-sharder daemon to automatically identify containers that are candidates for sharding and initiate the sharding process, instead of using the swift-manage-shard-ranges tool.

The container sharder uses an internal client and therefore requires an internal client configuration file to exist. By default the internal-client configuration file is expected to be found at */etc/swift/internal-client.conf*. An alternative location for the configuration file may be specified using the internal_client_conf_path option in the [container-sharder] config section.

The content of the internal-client configuration file should be the same as the *internal-client.conf-sample* file. In particular, the internal-client configuration should have:

```
account_autocreate = True
```

in the [proxy-server] section.

A container database may require several visits by the container-sharder daemon before it is fully sharded. On each visit the container-sharder daemon will move a subset of object records to new shard containers by cleaving new shard container databases from the original. By default, two shards are processed per visit; this number may be configured by the cleave_batch_size option.

The container-sharder daemon periodically writes progress data for containers that are being sharded to recon cache. For example:

```
"sharding_in_progress": {
    "all": [
        {
            "account": "AUTH_test",
            "active": 0,
            "cleaved": 2,
            "container": "c1",
            "created": 5,
            "db_state": "sharding",
            "error": null,
            "file_size": 26624,
            "found": 0,
            "meta_timestamp": "1525349617.46235",
            "node_index": 1,
            "object_count": 3349030,
            "path": <path_to_db>,
            "root": "AUTH_test/c1",
            "state": "sharding"
        }
    ]
}
```


This example indicates that from a total of 7 shard ranges, 2 have been cleaved whereas 5 remain in created state waiting to be cleaved.

Shard containers are created in an internal account and not visible to clients. By default, shard containers for an account `AUTH_test` are created in the internal account `.shards_AUTH_test`.

Once a container has started sharding, object updates to that container may be redirected to the shard container. The `container-sharder` daemon is also responsible for sending updates of a shards object count and bytes_used to the original container so that aggegrate object count and bytes used values can be returned in responses to client requests.

Note: The `container-sharder` daemon must continue to run on all container servers in order for shards object stats updates to be generated.

2.19.3 Under the hood

Terminology

Name	Description
Root container	The original container that lives in the users account. It holds references to its shard containers.
Retiring DB	The original database file that is to be sharded.
Fresh DB	A database file that will replace the retiring database.
Epoch	A timestamp at which the fresh DB is created; the epoch value is embedded in the fresh DB filename.
Shard range	A range of the object namespace defined by a lower bound and upper bound.
Shard container	A container that holds object records for a shard range. Shard containers exist in a hidden account mirroring the users account.
Parent container	The container from which a shard container has been cleaved. When first sharding a root container each shards parent container will be the root container. When sharding a shard container each shards parent container will be the sharding shard container.
Misplaced objects	Items that dont belong in a containers shard range. These will be moved to their correct location by the container-sharder.
Cleaving	The act of moving object records within a shard range to a shard container database.
Shrinking	The act of merging a small shard container into another shard container in order to delete the small shard container.
Donor	The shard range that is shrinking away.
Acceptor	The shard range into which a donor is merged.

Finding shard ranges

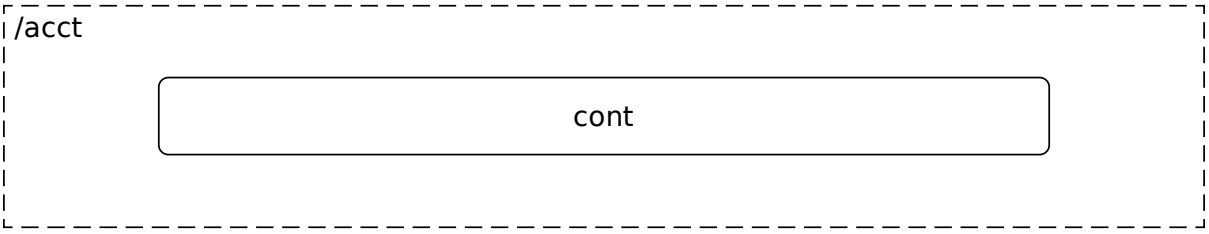
The end goal of sharding a container is to replace the original container database which has grown very large with a number of shard container databases, each of which is responsible for storing a range of the entire object namespace. The first step towards achieving this is to identify an appropriate set of contiguous object namespaces, known as shard ranges, each of which contains a similar sized portion of the containers current object content.

Shard ranges cannot simply be selected by sharding the namespace uniformly, because object names are not guaranteed to be distributed uniformly. If the container were naively sharded into two shard ranges, one containing all object names up to *m* and the other containing all object names beyond *m*, then if all object names actually start with *o* the outcome would be an extremely unbalanced pair of shard containers.

It is also too simplistic to assume that every container that requires sharding can be sharded into two. This might be the goal in the ideal world, but in practice there will be containers that have grown very large and should be sharded into many shards. Furthermore, the time required to find the exact mid-point of the existing object names in a large SQLite database would increase with container size.

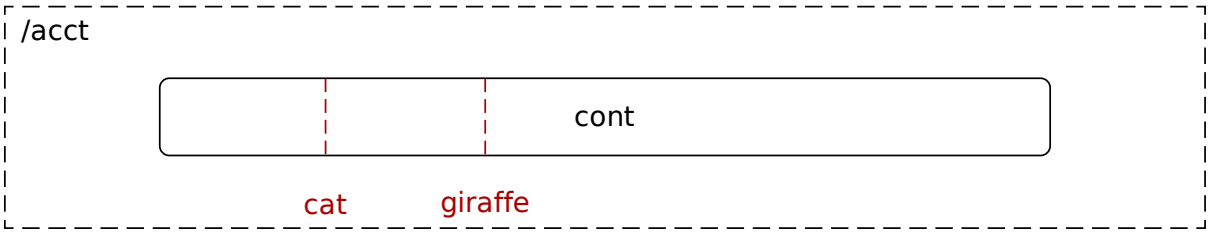
For these reasons, shard ranges of size *N* are found by searching for the *Nth* object in the database table, sorted by object name, and then searching for the *(2 * N)th* object, and so on until all objects have been searched. For a container that has exactly *2N* objects, the end result is the same as sharding the container at the midpoint of its object names. In practice sharding would typically be enabled for containers with great than *2N* objects and more than two shard ranges will be found, the last one probably containing less than *N* objects. With containers having large multiples of *N* objects, shard ranges can be identified in batches which enables more scalable solution.

To illustrate this process, consider a very large container in a user account `acct` that is a candidate for sharding:



The *swift-manage-shard-ranges CLI tool* tool `find` sub-command searches the object table for the *Nth* object whose name will become the upper bound of the first shard range, and the lower bound of the second shard range. The lower bound of the first shard range is the empty string.

For the purposes of this example the first upper bound is *cat*:



swift-manage-shard-ranges CLI tool continues to search the container to find further shard ranges, with the final upper bound also being the empty string.

Enabling sharding

Once shard ranges have been found the *swift-manage-shard-ranges CLI tool* `replace` sub-command is used to insert them into the *shard_ranges* table of the container database. In addition to its lower and upper bounds, each shard range is given a unique name.

The `enable` sub-command then creates some final state required to initiate sharding the container, including a special shard range record referred to as the containers *own_shard_range* whose name is equal to the containers path. This is used to keep a record of the object namespace that the container covers, which for user containers is always the entire namespace. Sharding of the container will only begin when its own shard ranges state has been set to `SHARDING`.

The ShardRange class

The *ShardRange* class provides methods for interactng with the attributes and state of a shard range. The class encapsulates the following properties:

- The name of the shard range which is also the name of the shard container used to hold object records in its namespace.
- Lower and upper bounds which define the object namespace of the shard range.
- A deleted flag.
- A timestamp at which the bounds and deleted flag were last modified.
- The object stats for the shard range i.e. object count and bytes used.
- A timestamp at which the object stats were last modified.
- The state of the shard range, and an epoch, which is the timestamp used in the shard containers database file name.
- A timestamp at which the state and epoch were last modified.

A shard range progresses through the following states:

- FOUND: the shard range has been identified in the container that is to be sharded but no resources have been created for it.
- CREATED: a shard container has been created to store the contents of the shard range.
- CLEAVED: the sharding containers contents for the shard range have been copied to the shard container from *at least one replica* of the sharding container.
- ACTIVE: a sharding containers constituent shard ranges are moved to this state when all shard ranges in the sharding container have been cleaved.
- SHRINKING: the shard range has been enabled for shrinking; or
- SHARDING: the shard range has been enabled for sharding into further sub-shards.
- SHARDED: the shard range has completed sharding or shrinking; the container will typically now have a number of constituent ACTIVE shard ranges.

Note: Shard range state represents the most advanced state of the shard range on any replica of the container. For example, a shard range in CLEAVED state may not have completed cleaving on all replicas but has cleaved on at least one replica.

Fresh and retiring database files

As alluded to earlier, writing to a large container causes increased latency for the container servers. Once sharding has been initiated on a container it is desirable to stop writing to the large database; ultimately it will be unlinked. This is primarily achieved by redirecting object updates to new shard containers as they are created (see *Redirecting object updates* below), but some object updates may still need to be accepted by the root container and other container metadata must still be modifiable.

To render the large *retiring* database effectively read-only, when the *container-sharder daemon* finds a container with a set of shard range records, including an *own_shard_range*, it first creates a fresh database file which will ultimately replace the existing *retiring* database. For a retiring DB whose filename is:

```
<hash>.db
```

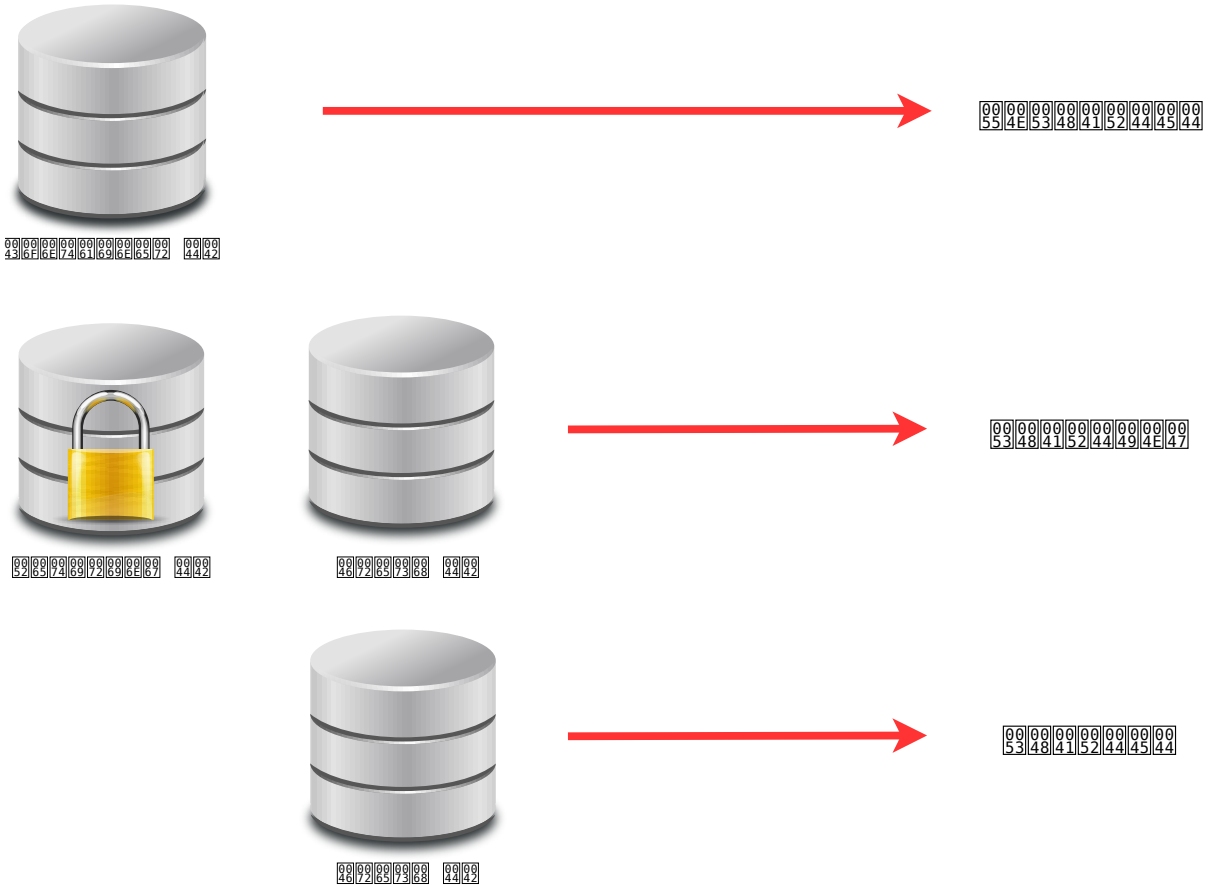
the fresh database file name is of the form:

```
<hash>_<epoch>.db
```

where *epoch* is a timestamp stored in the containers *own_shard_range*.

The fresh DB has a copy of the shard ranges table from the retiring DB and all other container metadata apart from the object records. Once a fresh DB file has been created it is used to store any new object updates and no more object records are written to the retiring DB file.

Once the sharding process has completed, the retiring DB file will be unlinked leaving only the fresh DB file in the containers directory. There are therefore three states that the container DB directory may be in during the sharding process: UNSHARDED, SHARDING and SHARDED.



If the container ever shrink to the point that is has no shards then the fresh DB starts to store object records, behaving the same as an unsharded container. This is known as the COLLAPSED state.

In summary, the DB states that any container replica may be in are:

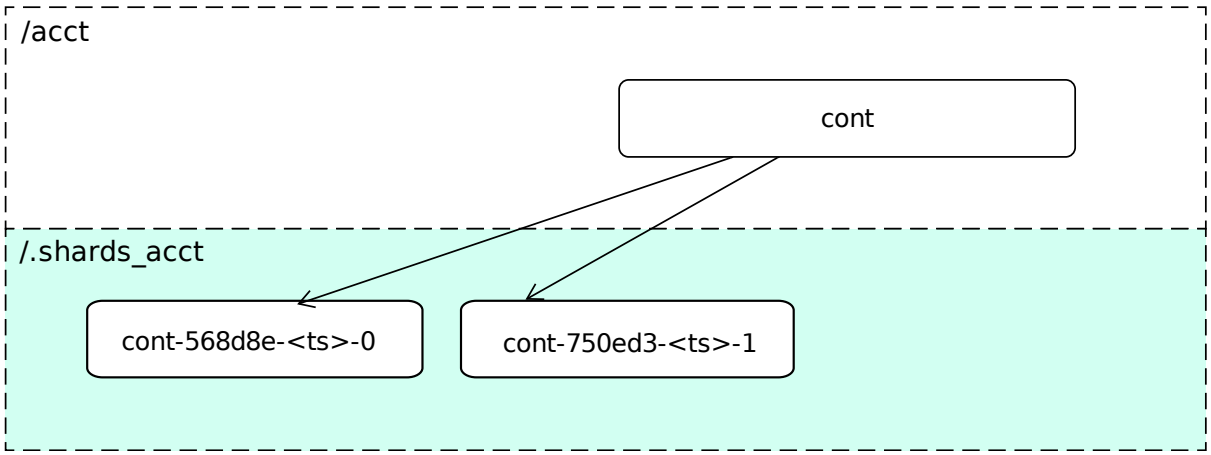
- UNSHARDED - In this state there is just one standard container database. All containers are originally in this state.
- SHARDING - There are now two databases, the retiring database and a fresh database. The fresh database stores any metadata, container level stats, an object holding table, and a table that stores shard ranges.

- SHARDED - There is only one database, the fresh database, which has one or more shard ranges in addition to its own shard range. The retiring database has been unlinked.
- COLLAPSED - There is only one database, the fresh database, which has only its own shard range and store object records.

Note: DB state is unique to each replica of a container and is not necessarily synchronised with shard range state.

Creating shard containers

The *container-sharder daemon* next creates a shard container for each shard range using the shard range name as the name of the shard container:



Each shard container has an *own_shard_range* record which has the lower and upper bounds of the object namespace for which it is responsible, and a reference to the sharding user container, which is referred to as the *root_container*. Unlike the *root_container*, the shard containers *own_shard_range* does not cover the entire namepsace.

A shard range name takes the form <shard_a>/<shard_c> where <*shard_a*> is a hidden account and <*shard_c*> is a container name that is derived from the root container.

The account name <*shard_a*> used for shard containers is formed by prefixing the user account with the string `.shards_`. This avoids namespace collisions and also keeps all the shard containers out of view from users of the account.

The container name for each shard container has the form:

```
<root container name>-<hash of parent container>-<timestamp>-<shard index>
```

where *root container name* is the name of the user container to which the contents of the shard container belong, *parent container* is the name of the container from which the shard is being cleaved, *timestamp* is the time at which the shard range was created and *shard index* is the position of the shard range in the name-ordered list of shard ranges for the *parent container*.

When sharding a user container the parent container name will be the same as the root container. However, if a *shard container* grows to a size that it requires sharding, then the parent container name for its shards will be the name of the sharding shard container.

For example, consider a user container with path `AUTH_user/c` which is sharded into two shard containers whose name will be:

```
.shards_AUTH_user/c-<hash(c)>-1234512345.12345-0
.shards_AUTH_user/c-<hash(c)>-1234512345.12345-1
```

If the first shard container is subsequently sharded into a further two shard containers then they will be named:

```
.shards_AUTH_user/c-<hash(c-<hash(c)>-1234567890.12345-0)>-1234567890.12345-0
.shards_AUTH_user/c-<hash(c-<hash(c)>-1234567890.12345-0)>-1234567890.12345-1
```

This naming scheme guarantees that shards, and shards of shards, each have a unique name of bounded length.

Cleaving shard containers

Having created empty shard containers the sharder daemon will proceed to cleave objects from the retiring database to each shard range. Cleaving occurs in batches of two (by default) shard ranges, so if a container has more than two shard ranges then the daemon must visit it multiple times to complete cleaving.

To cleave a shard range the daemon creates a shard database for the shard container on a local device. This device may be one of the shard containers primary nodes but often it will not. Object records from the corresponding shard range namespace are then copied from the retiring DB to this shard DB.

Swifts container replication mechanism is then used to replicate the shard DB to its primary nodes. Checks are made to ensure that the new shard container DB has been replicated to a sufficient number of its primary nodes before it is considered to have been successfully cleaved. By default the daemon requires successful replication of a new shard broker to at least a quorum of the container rings replica count, but this requirement can be tuned using the `shard_replication_quorum` option.

Once a shard range has been successfully cleaved from a retiring database the daemon transitions its state to `CLEAVED`. It should be noted that this state transition occurs as soon as any one of the retiring DB replicas has cleaved the shard range, and therefore does not imply that all retiring DB replicas have cleaved that range. The significance of the state transition is that the shard container is now considered suitable for contributing to object listings, since its contents are present on a quorum of its primary nodes and are the same as at least one of the retiring DBs for that namespace.

Once a shard range is in the `CLEAVED` state, the requirement for successful cleaving of other instances of the retirng DB may optionally be relaxed since it is not so imperative that their contents are replicated *immediately* to their primary nodes. The `existing_shard_replication_quorum` option can be used to reduce the quorum required for a cleaved shard range to be considered successfully replicated by the sharder daemon.

Note: Once cleaved, shard container DBs will continue to be replicated by the normal *container-replicator* daemon so that they will eventually be fully replicated to all primary nodes regardless of any replication quorum options used by the sharder daemon.

The cleaving progress of each replica of a retiring DB must be tracked independently of the shard range state. This is done using a per-DB CleavingContext object that maintains a cleaving cursor for the retiring DB that it is associated with. The cleaving cursor is simply the upper bound of the last shard range to have been cleaved *from that particular retiring DB*.

Each CleavingContext is stored in the sharding containers sysmeta under a key that is the `id` of the retiring DB. Since all container DB files have a unique `id`, this guarantees that each retiring DB will have a unique CleavingContext. Furthermore, if the retiring DB file is changed, for example by an rsync_then_merge replication operation which might change the contents of the DBs object table, then it will get a new unique CleavingContext.

A CleavingContext maintains other state that is used to ensure that a retiring DB is only considered to be fully cleaved, and ready to be deleted, if *all* of its object rows have been cleaved to a shard range.

Once all shard ranges have been cleaved from the retiring DB it is deleted. The container is now represented by the fresh DB which has a table of shard range records that point to the shard containers that store the containers object records.

Redirecting object updates

Once a shard container exists, object updates arising from new client requests and async pending files are directed to the shard container instead of the root container. This takes load off of the root container.

For a sharded (or partially sharded) container, when the proxy receives a new object request it issues a GET request to the container for data describing a shard container to which the object update should be sent. The proxy then annotates the object request with the shard container location so that the object server will forward object updates to the shard container. If those updates fail then the async pending file that is written on the object server contains the shard container location.

When the object updater processes async pending files for previously failed object updates, it may not find a shard container location. In this case the updater sends the update to the *root container*, which returns a redirection response with the shard container location.

Note: Object updates are directed to shard containers as soon as they exist, even if the retiring DB object records have not yet been cleaved to the shard container. This prevents further writes to the retiring DB and also avoids the fresh DB being polluted by new object updates. The goal is to ultimately have all object records in the shard containers and none in the root container.

Building container listings

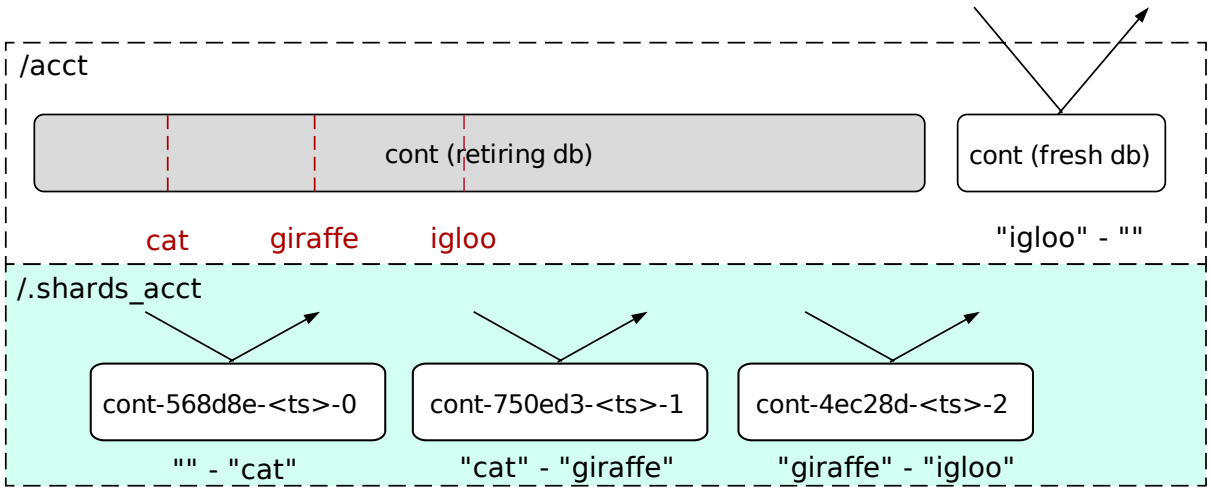
Listing requests for a sharded container are handled by querying the shard containers for components of the listing. The proxy forwards the client listing request to the root container, as it would for an unsharded container, but the container server responds with a list of shard ranges rather than objects. The proxy then queries each shard container in namespace order for their listing, until either the listing length limit is reached or all shard ranges have been listed.

While a container is still in the process of sharding, only *cleaved* shard ranges are used when building a container listing. Shard ranges that have not yet cleaved will not have any object records from the root container. The root container continues to provide listings for the uncleaved part of its namespace.

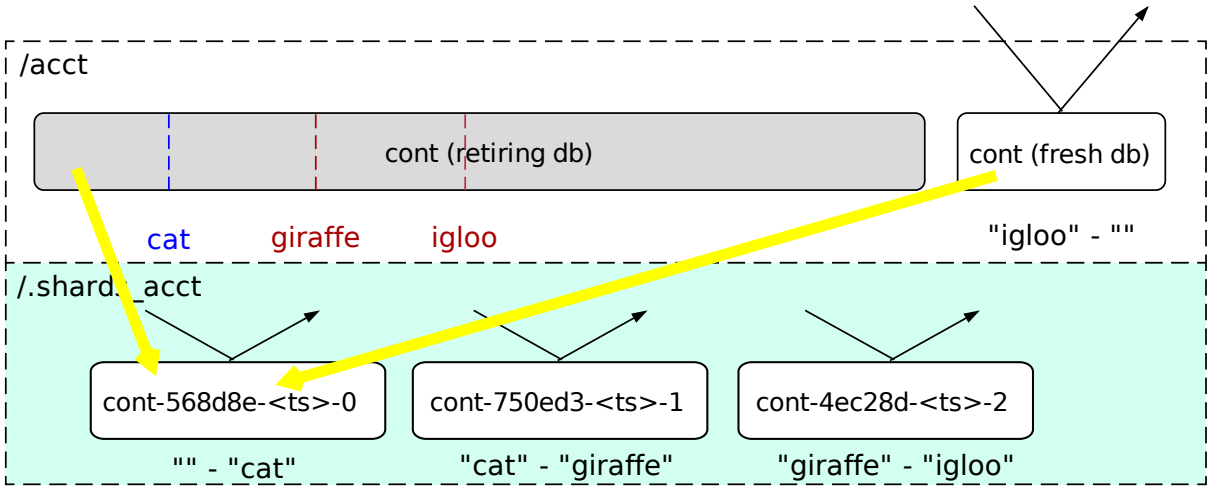
Note: New object updates are redirected to shard containers that have not yet been cleaved. These updates will not therefore be included in container listings until their shard range has been cleaved.

Example request redirection

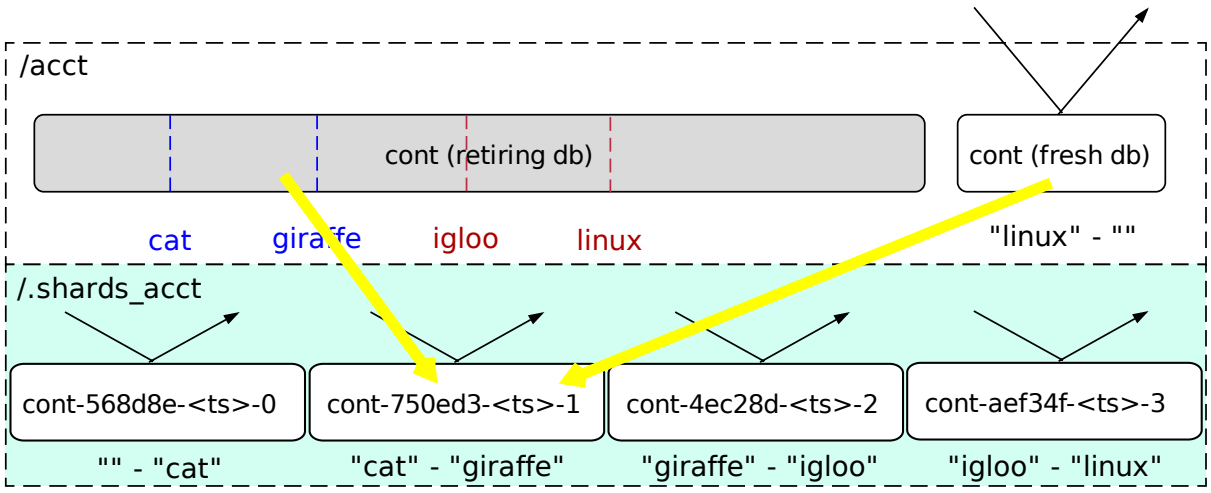
As an example, consider a sharding container in which 3 shard ranges have been found ending in cat, giraffe and igloo. Their respective shard containers have been created so update requests for objects up to igloo are redirected to the appropriate shard container. The root DB continues to handle listing requests and update requests for any object name beyond igloo.



The sharder daemon cleaves objects from the retiring DB to the shard range DBs; it also moves any misplaced objects from the root containers fresh DB to the shard DB. Cleaving progress is represented by the blue line. Once the first shard range has been cleaved listing requests for that namespace are directed to the shard container. The root container still provides listings for the remainder of the namespace.



The process continues: the sharder cleaves the next range and a new range is found with upper bound of linux. Now the root container only needs to handle listing requests up to giraffe and update requests for objects whose name is greater than linux. Load will continue to diminish on the root DB and be dispersed across the shard DBs.



Container replication

Shard range records are replicated between container DB replicas in much the same way as object records are for unsharded containers. However, the usual replication of object records between replicas of a container is halted as soon as a container is capable of being sharded. Instead, object records are moved to their new locations in shard containers. This avoids unnecessary replication traffic between container replicas.

To facilitate this, shard ranges are both pushed and pulled during replication, prior to any attempt to replicate objects. This means that the node initiating replication learns about shard ranges from the destination node early during the replication process and is able to skip object replication if it discovers that it has shard ranges and is able to shard.

Note: When the destination DB for container replication is missing then the complete_rsync replication mechanism is still used and in this case only both object records and shard range records are copied to the destination node.

Container deletion

Sharded containers may be deleted by a `DELETE` request just like an unsharded container. A sharded container must be empty before it can be deleted which implies that all of its shard containers must have reported that they are empty.

Shard containers are *not* immediately deleted when their root container is deleted; the shard containers remain undeleted so that they are able to continue to receive object updates that might arrive after the root container has been deleted. Shard containers continue to update their deleted root container with their object stats. If a shard container does receive object updates that cause it to no longer be empty then the root container will no longer be considered deleted once that shard container sends an object stats update.

Sharding a shard container

A shard container may grow to a size that requires it to be sharded. `swift-manage-shard-ranges` may be used to identify shard ranges within a shard container and enable sharding in the same way as for a root container. When a shard is sharding it notifies the root container of its shard ranges so that the root container can start to redirect object updates to the new sub-shards. When the shard has completed sharding the root is aware of all the new sub-shards and the sharding shard deletes its shard range record in the root container shard ranges table. At this point the root container is aware of all the new sub-shards which collectively cover the namespace of the now-deleted shard.

There is no hierarchy of shards beyond the root container and its immediate shards. When a shard shards, its sub-shards are effectively re-parented with the root container.

Shrinking a shard container

A shard containers contents may reduce to a point where the shard container is no longer required. If this happens then the shard container may be shrunk into another shard range. Shrinking is achieved in a similar way to sharding: an acceptor shard range is written to the shrinking shard containers shard ranges table; unlike sharding, where shard ranges each cover a subset of the sharding containers namespace, the acceptor shard range is a superset of the shrinking shard range.

Once given an acceptor shard range the shrinking shard will cleave itself to its acceptor, and then delete itself from the root container shard ranges table.

2.20 Building a Consistent Hashing Ring

2.20.1 Authored by Greg Holt, February 2011

This is a compilation of five posts I made earlier discussing how to build a consistent hashing ring. The posts seemed to be accessed quite frequently, so Ive gathered them all here on one page for easier reading.

Note: This is an historical document; as such, all code examples are Python 2. If this makes you squirm, think of it as pseudo-code. Regardless of implementation language, the state of the art in consistent-hashing and distributed systems more generally has advanced. We hope that this introduction from first principles will still prove informative, particularly with regard to how data is distributed within a Swift cluster.

Part 1

Consistent Hashing is a term used to describe a process where data is distributed using a hashing algorithm to determine its location. Using only the hash of the id of the data you can determine exactly where that data should be. This mapping of hashes to locations is usually termed a ring.

Probably the simplest hash is just a modulus of the id. For instance, if all ids are numbers and you have two machines you wish to distribute data to, you could just put all odd numbered ids on one machine and even numbered ids on the other. Assuming you have a balanced number of odd and even numbered ids, and a balanced data size per id, your data would be balanced between the two machines.

Since data ids are often textual names and not numbers, like paths for files or URLs, it makes sense to use a real hashing algorithm to convert the names to numbers first. Using MD5 for instance, the hash of the name mom.png is 4559a12e3e8da7c2186250c2f292e3af and the hash of dad.png is 096edcc4107e9e18d6a03a43b3853bea. Now, using the modulus, we can place mom.jpg on the odd machine and dad.png on the even one. Another benefit of using a hashing algorithm like MD5 is that the resulting hashes have a known even distribution, meaning your ids will be evenly distributed without worrying about keeping the id values themselves evenly distributed.

Here is a simple example of this in action:

```
from hashlib import md5
from struct import unpack_from

NODE_COUNT = 100
DATA_ID_COUNT = 10000000

node_counts = [0] * NODE_COUNT
for data_id in range(DATA_ID_COUNT):
    data_id = str(data_id)
    # This just pulls part of the hash out as an integer
    hsh = unpack_from('>I', md5(data_id).digest())[0]
    node_id = hsh % NODE_COUNT
    node_counts[node_id] += 1
desired_count = DATA_ID_COUNT / NODE_COUNT
print '%d: Desired data ids per node' % desired_count
max_count = max(node_counts)
over = 100.0 * (max_count - desired_count) / desired_count
print '%d: Most data ids on one node, %.02f%% over' % \
    (max_count, over)
min_count = min(node_counts)
under = 100.0 * (desired_count - min_count) / desired_count
print '%d: Least data ids on one node, %.02f%% under' % \
    (min_count, under)
```

```
100000: Desired data ids per node
100695: Most data ids on one node, 0.69% over
99073: Least data ids on one node, 0.93% under
```

So thats not bad at all; less than a percent over/under for distribution per node. In the next part of this series well examine where modulus distribution causes problems and how to improve our ring to overcome them.

Part 2

In Part 1 of this series, we did a simple test of using the modulus of a hash to locate data. We saw very good distribution, but thats only part of the story. Distributed systems not only need to distribute load, but they often also need to grow as more and more data is placed in it.

So lets imagine we have a 100 node system up and running using our previous algorithm, but its starting to get full so we want to add another node. When we add that 101st node to our algorithm we notice that many ids now map to different nodes than they previously did. Were going to have to shuffle a ton of data around our system to get it all into place again.

Lets examine whats happened on a much smaller scale: just 2 nodes again, node 0 gets even ids and node 1 gets odd ids. So data id 100 would map to node 0, data id 101 to node 1, data id 102 to node 0, etc. This is simply node = id % 2. Now we add a third node (node 2) for more space, so we want node = id % 3. So now data id 100 maps to node id 1, data id 101 to node 2, and data id 102 to node 0. So we have to move data for 2 of our 3 ids so they can be found again.

Lets examine this at a larger scale:

```
from hashlib import md5
from struct import unpack_from

NODE_COUNT = 100
NEW_NODE_COUNT = 101
DATA_ID_COUNT = 10000000

moved_ids = 0
for data_id in range(DATA_ID_COUNT):
    data_id = str(data_id)
    hsh = unpack_from('>I', md5(str(data_id)).digest())[0]
    node_id = hsh % NODE_COUNT
    new_node_id = hsh % NEW_NODE_COUNT
    if node_id != new_node_id:
        moved_ids += 1
percent_moved = 100.0 * moved_ids / DATA_ID_COUNT
print '%d ids moved, %.02f%%' % (moved_ids, percent_moved)
```

```
9900989 ids moved, 99.01%
```

Wow, thats severe. Wed have to shuffle around 99% of our data just to increase our capacity 1%! We need a new algorithm that combats this behavior.

This is where the ring really comes in. We can assign ranges of hashes directly to nodes and then use an algorithm that minimizes the changes to those ranges. Back to our small scale, lets say our ids range from 0 to 999. We have two nodes and well assign data ids 0499 to node 0 and 500999 to node 1. Later, when we add node 2, we can take half the data ids from node 0 and half from node 1, minimizing the amount of data that needs to move.

Lets examine this at a larger scale:

```
from bisect import bisect_left
from hashlib import md5
from struct import unpack_from
```

(continues on next page)

(continued from previous page)

```
NODE_COUNT = 100
NEW_NODE_COUNT = 101
DATA_ID_COUNT = 10000000

node_range_starts = []
for node_id in range(NODE_COUNT):
    node_range_starts.append(DATA_ID_COUNT /
                             NODE_COUNT * node_id)
new_node_range_starts = []
for new_node_id in range(NEW_NODE_COUNT):
    new_node_range_starts.append(DATA_ID_COUNT /
                             NEW_NODE_COUNT * new_node_id)
moved_ids = 0
for data_id in range(DATA_ID_COUNT):
    data_id = str(data_id)
    hsh = unpack_from('>I', md5(str(data_id)).digest())[0]
    node_id = bisect_left(node_range_starts,
                          hsh % DATA_ID_COUNT) % NODE_COUNT
    new_node_id = bisect_left(new_node_range_starts,
                          hsh % DATA_ID_COUNT) % NEW_NODE_COUNT
    if node_id != new_node_id:
        moved_ids += 1
percent_moved = 100.0 * moved_ids / DATA_ID_COUNT
print '%d ids moved, %.02f%%' % (moved_ids, percent_moved)
```

```
4901707 ids moved, 49.02%
```

Okay, that is better. But still, moving 50% of our data to add 1% capacity is not very good. If we examine what happened more closely well see what is an accordion effect. We shrunk node 0s range a bit to give to the new node, but that shifted all the other nodes ranges by the same amount.

We can minimize the change to a nodes assigned range by assigning several smaller ranges instead of the single broad range we were before. This can be done by creating virtual nodes for each node. So 100 nodes might have 1000 virtual nodes. Lets examine how that might work.

```
from bisect import bisect_left
from hashlib import md5
from struct import unpack_from

NODE_COUNT = 100
DATA_ID_COUNT = 10000000
VNODE_COUNT = 1000

vnode_range_starts = []
vnode2node = []
for vnode_id in range(VNODE_COUNT):
    vnode_range_starts.append(DATA_ID_COUNT /
                              VNODE_COUNT * vnode_id)
    vnode2node.append(vnode_id % NODE_COUNT)
```

(continues on next page)

(continued from previous page)

```
new_vnode2node = list(vnode2node)
new_node_id = NODE_COUNT
NEW_NODE_COUNT = NODE_COUNT + 1
vnodes_to_reassign = VNODE_COUNT / NEW_NODE_COUNT
while vnodes_to_reassign > 0:
    for node_to_take_from in range(NODE_COUNT):
        for vnode_id, node_id in enumerate(new_vnode2node):
            if node_id == node_to_take_from:
                new_vnode2node[vnode_id] = new_node_id
                vnodes_to_reassign -= 1
                break
        if vnodes_to_reassign <= 0:
            break
moved_ids = 0
for data_id in range(DATA_ID_COUNT):
    data_id = str(data_id)
    hsh = unpack_from('>I', md5(str(data_id)).digest())[0]
    vnode_id = bisect_left(vnode_range_starts,
                           hsh % DATA_ID_COUNT) % VNODE_COUNT
    node_id = vnode2node[vnode_id]
    new_node_id = new_vnode2node[vnode_id]
    if node_id != new_node_id:
        moved_ids += 1
percent_moved = 100.0 * moved_ids / DATA_ID_COUNT
print '%d ids moved, %.02f%%' % (moved_ids, percent_moved)
```

```
90423 ids moved, 0.90%
```

There we go, we added 1% capacity and only moved 0.9% of existing data. The vnode_range_starts list seems a bit out of place though. Its values are calculated and never change for the lifetime of the cluster, so lets optimize that out.

```
from bisect import bisect_left
from hashlib import md5
from struct import unpack_from

NODE_COUNT = 100
DATA_ID_COUNT = 10000000
VNODE_COUNT = 1000

vnode2node = []
for vnode_id in range(VNODE_COUNT):
    vnode2node.append(vnode_id % NODE_COUNT)
new_vnode2node = list(vnode2node)
new_node_id = NODE_COUNT
vnodes_to_reassign = VNODE_COUNT / (NODE_COUNT + 1)
while vnodes_to_reassign > 0:
    for node_to_take_from in range(NODE_COUNT):
        for vnode_id, node_id in enumerate(vnode2node):
```

(continues on next page)

(continued from previous page)

```
            if node_id == node_to_take_from:
                vnode2node[vnode_id] = new_node_id
                vnodes_to_reassign -= 1
                break
        if vnodes_to_reassign <= 0:
            break
moved_ids = 0
for data_id in range(DATA_ID_COUNT):
    data_id = str(data_id)
    hsh = unpack_from('>I', md5(str(data_id)).digest())[0]
    vnode_id = hsh % VNODE_COUNT
    node_id = vnode2node[vnode_id]
    new_node_id = new_vnode2node[vnode_id]
    if node_id != new_node_id:
        moved_ids += 1
percent_moved = 100.0 * moved_ids / DATA_ID_COUNT
print '%d ids moved, %.02f%%' % (moved_ids, percent_moved)
```

```
89841 ids moved, 0.90%
```

There we go. In the next part of this series, will further examine the algorithms limitations and how to improve on it.

Part 3

In Part 2 of this series, we reached an algorithm that performed well even when adding new nodes to the cluster. We used 1000 virtual nodes that could be independently assigned to nodes, allowing us to minimize the amount of data moved when a node was added.

The number of virtual nodes puts a cap on how many real nodes you can have. For example, if you have 1000 virtual nodes and you try to add a 1001st real node, you cant assign a virtual node to it without leaving another real node with no assignment, leaving you with just 1000 active real nodes still.

Unfortunately, the number of virtual nodes created at the beginning can never change for the life of the cluster without a lot of careful work. For example, you could double the virtual node count by splitting each existing virtual node in half and assigning both halves to the same real node. However, if the real node uses the virtual nodes id to optimally store the data (for example, all data might be stored in /[virtual node id]/[data id]) it would have to move data around locally to reflect the change. And it would have to resolve data using both the new and old locations while the moves were taking place, making atomic operations difficult or impossible.

Lets continue with this assumption that changing the virtual node count is more work than its worth, but keep in mind that some applications might be fine with this.

The easiest way to deal with this limitation is to make the limit high enough that it wont matter. For instance, if we decide our cluster will never exceed 60,000 real nodes, we can just make 60,000 virtual nodes.

Also, we should include in our calculations the relative size of our nodes. For instance, a year from now we might have real nodes that can handle twice the capacity of our current nodes. So wed want to assign twice the virtual nodes to those future nodes, so maybe we should raise our virtual node estimate to 120,000.

A good rule to follow might be to calculate 100 virtual nodes to each real node at maximum capacity. This would allow you to alter the load on any given node by 1%, even at max capacity, which is pretty fine tuning. So now were at 6,000,000 virtual nodes for a max capacity cluster of 60,000 real nodes.

6 million virtual nodes seems like a lot, and it might seem like wed use up way too much memory. But the only structure this affects is the virtual node to real node mapping. The base amount of memory required would be 6 million times 2 bytes (to store a real node id from 0 to 65,535). 12 megabytes of memory just isnt that much to use these days.

Even with all the overhead of flexible data types, things arent that bad. I changed the code from the previous part in this series to have 60,000 real and 6,000,000 virtual nodes, changed the list to an array(H), and python topped out at 27m of resident memory and that includes two rings.

To change terminology a bit, were going to start calling these virtual nodes partitions. This will make it a bit easier to discern between the two types of nodes weve been talking about so far. Also, it makes sense to talk about partitions as they are really just unchanging sections of the hash space.

Were also going to always keep the partition count a power of two. This makes it easy to just use bit manipulation on the hash to determine the partition rather than modulus. It isnt much faster, but it is a little. So, heres our updated ring code, using 8,388,608 (2 ** 23) partitions and 65,536 nodes. Weve upped the sample data id set and checked the distribution to make sure we havent broken anything.

```
from array import array
from hashlib import md5
from struct import unpack_from

PARTITION_POWER = 23
PARTITION_SHIFT = 32 - PARTITION_POWER
NODE_COUNT = 65536
DATA_ID_COUNT = 100000000

part2node = array('H')
for part in range(2 ** PARTITION_POWER):
    part2node.append(part % NODE_COUNT)
node_counts = [0] * NODE_COUNT
for data_id in range(DATA_ID_COUNT):
    data_id = str(data_id)
    part = unpack_from('>I',
        md5(str(data_id)).digest())[0] >> PARTITION_SHIFT
    node_id = part2node[part]
    node_counts[node_id] += 1
desired_count = DATA_ID_COUNT / NODE_COUNT
print '%d: Desired data ids per node' % desired_count
max_count = max(node_counts)
over = 100.0 * (max_count - desired_count) / desired_count
print '%d: Most data ids on one node, %.02f%% over' % \
    (max_count, over)
min_count = min(node_counts)
under = 100.0 * (desired_count - min_count) / desired_count
print '%d: Least data ids on one node, %.02f%% under' % \
    (min_count, under)
```

```
1525: Desired data ids per node
1683: Most data ids on one node, 10.36% over
1360: Least data ids on one node, 10.82% under
```

Hmm. +10% seems a bit high, but I reran with 65,536 partitions and 256 nodes and got +0.4% so its just that our sample size (100m) is too small for our number of partitions (8m). Itll take way too long to run experiments with an even larger sample size, so lets reduce back down to these lesser numbers. (To be certain, I reran at the full version with a 10 billion data id sample set and got +1%, but it took 6.5 hours to run.)

In the next part of this series, well talk about how to increase the durability of our data in the cluster.

Part 4

In Part 3 of this series, we just further discussed partitions (virtual nodes) and cleaned up our code a bit based on that. Now, lets talk about how to increase the durability and availability of our data in the cluster.

For many distributed data stores, durability is quite important. Either RAID arrays or individually distinct copies of data are required. While RAID will increase the durability, it does nothing to increase the availability if the RAID machine crashes, the data may be safe but inaccessible until repairs are done. If we keep distinct copies of the data on different machines and a machine crashes, the other copies will still be available while we repair the broken machine.

An easy way to gain this multiple copy durability/availability is to just use multiple rings and groups of nodes. For instance, to achieve the industry standard of three copies, youd split the nodes into three groups and each group would have its own ring and each would receive a copy of each data item. This can work well enough, but has the drawback that expanding capacity requires adding three nodes at a time and that losing one node essentially lowers capacity by three times that nodes capacity.

Instead, lets use a different, but common, approach of meeting our requirements with a single ring. This can be done by walking the ring from the starting point and looking for additional distinct nodes. Heres code that supports a variable number of replicas (set to 3 for testing):

```
from array import array
from hashlib import md5
from struct import unpack_from

REPLICAS = 3
PARTITION_POWER = 16
PARTITION_SHIFT = 32 - PARTITION_POWER
PARTITION_MAX = 2 ** PARTITION_POWER - 1
NODE_COUNT = 256
DATA_ID_COUNT = 10000000

part2node = array('H')
for part in range(2 ** PARTITION_POWER):
    part2node.append(part % NODE_COUNT)
node_counts = [0] * NODE_COUNT
for data_id in range(DATA_ID_COUNT):
    data_id = str(data_id)
```

(continues on next page)

(continued from previous page)

```
    part = unpack_from('>I',
        md5(str(data_id)).digest())[0] >> PARTITION_SHIFT
    node_ids = [part2node[part]]
    node_counts[node_ids[0]] += 1
    for replica in range(1, REPLICAS):
        while part2node[part] in node_ids:
            part += 1
            if part > PARTITION_MAX:
                part = 0
        node_ids.append(part2node[part])
        node_counts[node_ids[-1]] += 1
desired_count = DATA_ID_COUNT / NODE_COUNT * REPLICAS
print '%d: Desired data ids per node' % desired_count
max_count = max(node_counts)
over = 100.0 * (max_count - desired_count) / desired_count
print '%d: Most data ids on one node, %.02f%% over' % \
    (max_count, over)
min_count = min(node_counts)
under = 100.0 * (desired_count - min_count) / desired_count
print '%d: Least data ids on one node, %.02f%% under' % \
    (min_count, under)
```

```
117186: Desired data ids per node
118133: Most data ids on one node, 0.81% over
116093: Least data ids on one node, 0.93% under
```

Thats pretty good; less than 1% over/under. While this works well, there are a couple of problems.

First, because of how weve initially assigned the partitions to nodes, all the partitions for a given node have their extra copies on the same other two nodes. The problem here is that when a machine fails, the load on these other nodes will jump by that amount. Itd be better if we initially shuffled the partition assignment to distribute the failover load better.

The other problem is a bit harder to explain, but deals with physical separation of machines. Imagine you can only put 16 machines in a rack in your datacenter. The 256 nodes weve been using would fill 16 racks. With our current code, if a rack goes out (power problem, network issue, etc.) there is a good chance some data will have all three copies in that rack, becoming inaccessible. We can fix this shortcoming by adding the concept of zones to our nodes, and then ensuring that replicas are stored in distinct zones.

```
from array import array
from hashlib import md5
from random import shuffle
from struct import unpack_from

REPLICAS = 3
PARTITION_POWER = 16
PARTITION_SHIFT = 32 - PARTITION_POWER
PARTITION_MAX = 2 ** PARTITION_POWER - 1
NODE_COUNT = 256
ZONE_COUNT = 16
```

(continues on next page)

(continued from previous page)

```
DATA_ID_COUNT = 10000000

node2zone = []
while len(node2zone) < NODE_COUNT:
    zone = 0
    while zone < ZONE_COUNT and len(node2zone) < NODE_COUNT:
        node2zone.append(zone)
        zone += 1
part2node = array('H')
for part in range(2 ** PARTITION_POWER):
    part2node.append(part % NODE_COUNT)
shuffle(part2node)
node_counts = [0] * NODE_COUNT
zone_counts = [0] * ZONE_COUNT
for data_id in range(DATA_ID_COUNT):
    data_id = str(data_id)
    part = unpack_from('>I',
        md5(str(data_id)).digest())[0] >> PARTITION_SHIFT
    node_ids = [part2node[part]]
    zones = [node2zone[node_ids[0]]]
    node_counts[node_ids[0]] += 1
    zone_counts[zones[0]] += 1
    for replica in range(1, REPLICAS):
        while part2node[part] in node_ids and \
                node2zone[part2node[part]] in zones:
            part += 1
            if part > PARTITION_MAX:
                part = 0
        node_ids.append(part2node[part])
        zones.append(node2zone[node_ids[-1]])
        node_counts[node_ids[-1]] += 1
        zone_counts[zones[-1]] += 1
desired_count = DATA_ID_COUNT / NODE_COUNT * REPLICAS
print '%d: Desired data ids per node' % desired_count
max_count = max(node_counts)
over = 100.0 * (max_count - desired_count) / desired_count
print '%d: Most data ids on one node, %.02f%% over' % \
    (max_count, over)
min_count = min(node_counts)
under = 100.0 * (desired_count - min_count) / desired_count
print '%d: Least data ids on one node, %.02f%% under' % \
    (min_count, under)
desired_count = DATA_ID_COUNT / ZONE_COUNT * REPLICAS
print '%d: Desired data ids per zone' % desired_count
max_count = max(zone_counts)
over = 100.0 * (max_count - desired_count) / desired_count
print '%d: Most data ids in one zone, %.02f%% over' % \
    (max_count, over)
min_count = min(zone_counts)
```

(continues on next page)

(continued from previous page)

```
under = 100.0 * (desired_count - min_count) / desired_count
print '%d: Least data ids in one zone, %.02f%% under' % \
    (min_count, under)
```

```
117186: Desired data ids per node
118782: Most data ids on one node, 1.36% over
115632: Least data ids on one node, 1.33% under
1875000: Desired data ids per zone
1878533: Most data ids in one zone, 0.19% over
1869070: Least data ids in one zone, 0.32% under
```

So the shuffle and zone distinctions affected our distribution some, but still definitely good enough. This test took about 64 seconds to run on my machine.

Theres a completely alternate, and quite common, way of accomplishing these same requirements. This alternate method doesnt use partitions at all, but instead just assigns anchors to the nodes within the hash space. Finding the first node for a given hash just involves walking this anchor ring for the next node, and finding additional nodes works similarly as before. To attain the equivalent of our virtual nodes, each real node is assigned multiple anchors.

```
from bisect import bisect_left
from hashlib import md5
from struct import unpack_from

REPLICAS = 3
NODE_COUNT = 256
ZONE_COUNT = 16
DATA_ID_COUNT = 10000000
VNODE_COUNT = 100

node2zone = []
while len(node2zone) < NODE_COUNT:
    zone = 0
    while zone < ZONE_COUNT and len(node2zone) < NODE_COUNT:
        node2zone.append(zone)
        zone += 1
hash2index = []
index2node = []
for node in range(NODE_COUNT):
    for vnode in range(VNODE_COUNT):
        hsh = unpack_from('>I', md5(str(node)).digest())[0]
        index = bisect_left(hash2index, hsh)
        if index > len(hash2index):
            index = 0
        hash2index.insert(index, hsh)
        index2node.insert(index, node)
node_counts = [0] * NODE_COUNT
zone_counts = [0] * ZONE_COUNT
for data_id in range(DATA_ID_COUNT):
    data_id = str(data_id)
```

(continues on next page)

(continued from previous page)

```
    hsh = unpack_from('>I', md5(str(data_id)).digest())[0]
    index = bisect_left(hash2index, hsh)
    if index >= len(hash2index):
        index = 0
    node_ids = [index2node[index]]
    zones = [node2zone[node_ids[0]]]
    node_counts[node_ids[0]] += 1
    zone_counts[zones[0]] += 1
    for replica in range(1, REPLICAS):
        while index2node[index] in node_ids and \
                node2zone[index2node[index]] in zones:
            index += 1
            if index >= len(hash2index):
                index = 0
        node_ids.append(index2node[index])
        zones.append(node2zone[node_ids[-1]])
        node_counts[node_ids[-1]] += 1
        zone_counts[zones[-1]] += 1
desired_count = DATA_ID_COUNT / NODE_COUNT * REPLICAS
print '%d: Desired data ids per node' % desired_count
max_count = max(node_counts)
over = 100.0 * (max_count - desired_count) / desired_count
print '%d: Most data ids on one node, %.02f%% over' % \
    (max_count, over)
min_count = min(node_counts)
under = 100.0 * (desired_count - min_count) / desired_count
print '%d: Least data ids on one node, %.02f%% under' % \
    (min_count, under)
desired_count = DATA_ID_COUNT / ZONE_COUNT * REPLICAS
print '%d: Desired data ids per zone' % desired_count
max_count = max(zone_counts)
over = 100.0 * (max_count - desired_count) / desired_count
print '%d: Most data ids in one zone, %.02f%% over' % \
    (max_count, over)
min_count = min(zone_counts)
under = 100.0 * (desired_count - min_count) / desired_count
print '%d: Least data ids in one zone, %.02f%% under' % \
    (min_count, under)
```

```
117186: Desired data ids per node
351282: Most data ids on one node, 199.76% over
15965: Least data ids on one node, 86.38% under
1875000: Desired data ids per zone
2248496: Most data ids in one zone, 19.92% over
1378013: Least data ids in one zone, 26.51% under
```

This test took over 15 minutes to run! Unfortunately, this method also gives much less control over the distribution. To get better distribution, you have to add more virtual nodes, which eats up more memory and takes even more time to build the ring and perform distinct node lookups. The most common operation, data id lookup, can be improved (by predetermining each virtual nodes failover nodes, for

instance) but it starts off so far behind our first approach that well just stick with that.

In the next part of this series, well start to wrap all this up into a useful Python module.

Part 5

In Part 4 of this series, we ended up with a multiple copy, distinctly zoned ring. Or at least the start of it. In this final part well package the code up into a useable Python module and then add one last feature. First, lets separate the ring itself from the building of the data for the ring and its testing.

```
from array import array
from hashlib import md5
from random import shuffle
from struct import unpack_from
from time import time

class Ring(object):

    def __init__(self, nodes, part2node, replicas):
        self.nodes = nodes
        self.part2node = part2node
        self.replicas = replicas
        partition_power = 1
        while 2 ** partition_power < len(part2node):
            partition_power += 1
        if len(part2node) != 2 ** partition_power:
            raise Exception("part2node's length is not an "
                            "exact power of 2")
        self.partition_shift = 32 - partition_power

    def get_nodes(self, data_id):
        data_id = str(data_id)
        part = unpack_from('>I',
           md5(data_id).digest())[0] >> self.partition_shift
        node_ids = [self.part2node[part]]
        zones = [self.nodes[node_ids[0]]]
        for replica in range(1, self.replicas):
            while self.part2node[part] in node_ids and \
                    self.nodes[self.part2node[part]] in zones:
                part += 1
                if part >= len(self.part2node):
                    part = 0
            node_ids.append(self.part2node[part])
            zones.append(self.nodes[node_ids[-1]])
        return [self.nodes[n] for n in node_ids]

def build_ring(nodes, partition_power, replicas):
    begin = time()
    part2node = array('H')
    for part in range(2 ** partition_power):
        part2node.append(part % len(nodes))
```

(continues on next page)

(continued from previous page)

```
    shuffle(part2node)
    ring = Ring(nodes, part2node, replicas)
    print '%.02fs to build ring' % (time() - begin)
    return ring

def test_ring(ring):
    begin = time()
    DATA_ID_COUNT = 10000000
    node_counts = {}
    zone_counts = {}
    for data_id in range(DATA_ID_COUNT):
        for node in ring.get_nodes(data_id):
            node_counts[node['id']] = \
                node_counts.get(node['id'], 0) + 1
            zone_counts[node['zone']] = \
                zone_counts.get(node['zone'], 0) + 1
    print '%ds to test ring' % (time() - begin)
    desired_count = \
        DATA_ID_COUNT / len(ring.nodes) * REPLICAS
    print '%d: Desired data ids per node' % desired_count
    max_count = max(node_counts.values())
    over = \
        100.0 * (max_count - desired_count) / desired_count
    print '%d: Most data ids on one node, %.02f%% over' % \
        (max_count, over)
    min_count = min(node_counts.values())
    under = \
        100.0 * (desired_count - min_count) / desired_count
    print '%d: Least data ids on one node, %.02f%% under' % \
        (min_count, under)
    zone_count = \
        len(set(n['zone'] for n in ring.nodes.values()))
    desired_count = \
        DATA_ID_COUNT / zone_count * ring.replicas
    print '%d: Desired data ids per zone' % desired_count
    max_count = max(zone_counts.values())
    over = \
        100.0 * (max_count - desired_count) / desired_count
    print '%d: Most data ids in one zone, %.02f%% over' % \
        (max_count, over)
    min_count = min(zone_counts.values())
    under = \
        100.0 * (desired_count - min_count) / desired_count
    print '%d: Least data ids in one zone, %.02f%% under' % \
        (min_count, under)

if __name__ == '__main__':
    PARTITION_POWER = 16
    REPLICAS = 3
```

(continues on next page)

(continued from previous page)

```
    NODE_COUNT = 256
    ZONE_COUNT = 16
    nodes = {}
    while len(nodes) < NODE_COUNT:
        zone = 0
        while zone < ZONE_COUNT and len(nodes) < NODE_COUNT:
            node_id = len(nodes)
            nodes[node_id] = {'id': node_id, 'zone': zone}
            zone += 1
    ring = build_ring(nodes, PARTITION_POWER, REPLICAS)
    test_ring(ring)
```

```
0.06s to build ring
82s to test ring
117186: Desired data ids per node
118773: Most data ids on one node, 1.35% over
115801: Least data ids on one node, 1.18% under
1875000: Desired data ids per zone
1878339: Most data ids in one zone, 0.18% over
1869914: Least data ids in one zone, 0.27% under
```

It takes a bit longer to test our ring, but thats mostly because of the switch to dictionaries from arrays for various items. Having node dictionaries is nice because you can attach any node information you want directly there (ip addresses, tcp ports, drive paths, etc.). But were still on track for further testing; our distribution is still good.

Now, lets add our one last feature to our ring: the concept of weights. Weights are useful because the nodes you add later in a rings life are likely to have more capacity than those you have at the outset. For this test, well make half our nodes have twice the weight. Well have to change build_ring to give more partitions to the nodes with more weight and well change test_ring to take into account these weights. Since weve changed so much Ill just post the entire module again:

```
from array import array
from hashlib import md5
from random import shuffle
from struct import unpack_from
from time import time

class Ring(object):

    def __init__(self, nodes, part2node, replicas):
        self.nodes = nodes
        self.part2node = part2node
        self.replicas = replicas
        partition_power = 1
        while 2 ** partition_power < len(part2node):
            partition_power += 1
        if len(part2node) != 2 ** partition_power:
            raise Exception("part2node's length is not an "
                            "exact power of 2")
```

(continues on next page)

(continued from previous page)

```
        self.partition_shift = 32 - partition_power

    def get_nodes(self, data_id):
        data_id = str(data_id)
        part = unpack_from('>I',
           md5(data_id).digest())[0] >> self.partition_shift
        node_ids = [self.part2node[part]]
        zones = [self.nodes[node_ids[0]]]
        for replica in range(1, self.replicas):
            while self.part2node[part] in node_ids and \
                  self.nodes[self.part2node[part]] in zones:
                part += 1
                if part >= len(self.part2node):
                    part = 0
            node_ids.append(self.part2node[part])
            zones.append(self.nodes[node_ids[-1]])
        return [self.nodes[n] for n in node_ids]

def build_ring(nodes, partition_power, replicas):
    begin = time()
    parts = 2 ** partition_power
    total_weight = \
        float(sum(n['weight'] for n in nodes.values()))
    for node in nodes.values():
        node['desired_parts'] = \
            parts / total_weight * node['weight']
    part2node = array('H')
    for part in range(2 ** partition_power):
        for node in nodes.values():
            if node['desired_parts'] >= 1:
                node['desired_parts'] -= 1
                part2node.append(node['id'])
                break
        else:
            for node in nodes.values():
                if node['desired_parts'] >= 0:
                    node['desired_parts'] -= 1
                    part2node.append(node['id'])
                    break
    shuffle(part2node)
    ring = Ring(nodes, part2node, replicas)
    print '%.02fs to build ring' % (time() - begin)
    return ring

def test_ring(ring):
    begin = time()
    DATA_ID_COUNT = 10000000
    node_counts = {}
    zone_counts = {}
```

(continues on next page)

(continued from previous page)

```
    for data_id in range(DATA_ID_COUNT):
        for node in ring.get_nodes(data_id):
            node_counts[node['id']] = \
                node_counts.get(node['id'], 0) + 1
            zone_counts[node['zone']] = \
                zone_counts.get(node['zone'], 0) + 1
    print '%ds to test ring' % (time() - begin)
    total_weight = float(sum(n['weight'] for n in
                             ring.nodes.values()))
    max_over = 0
    max_under = 0
    for node in ring.nodes.values():
        desired = DATA_ID_COUNT * REPLICAS * \
            node['weight'] / total_weight
        diff = node_counts[node['id']] - desired
        if diff > 0:
            over = 100.0 * diff / desired
            if over > max_over:
                max_over = over
        else:
            under = 100.0 * (-diff) / desired
            if under > max_under:
                max_under = under
    print '%.02f%% max node over' % max_over
    print '%.02f%% max node under' % max_under
    max_over = 0
    max_under = 0
    for zone in set(n['zone'] for n in
                    ring.nodes.values()):
        zone_weight = sum(n['weight'] for n in
            ring.nodes.values() if n['zone'] == zone)
        desired = DATA_ID_COUNT * REPLICAS * \
            zone_weight / total_weight
        diff = zone_counts[zone] - desired
        if diff > 0:
            over = 100.0 * diff / desired
            if over > max_over:
                max_over = over
        else:
            under = 100.0 * (-diff) / desired
            if under > max_under:
                max_under = under
    print '%.02f%% max zone over' % max_over
    print '%.02f%% max zone under' % max_under

if __name__ == '__main__':
    PARTITION_POWER = 16
    REPLICAS = 3
    NODE_COUNT = 256
```

(continues on next page)

(continued from previous page)

```
    ZONE_COUNT = 16
    nodes = {}
    while len(nodes) < NODE_COUNT:
        zone = 0
        while zone < ZONE_COUNT and len(nodes) < NODE_COUNT:
            node_id = len(nodes)
            nodes[node_id] = {'id': node_id, 'zone': zone,
                              'weight': 1.0 + (node_id % 2)}
            zone += 1
    ring = build_ring(nodes, PARTITION_POWER, REPLICAS)
    test_ring(ring)
```

```
0.88s to build ring
86s to test ring
1.66% max over
1.46% max under
0.28% max zone over
0.23% max zone under
```

So things are still good, even though we have differently weighted nodes. I ran another test with this code using random weights from 1 to 100 and got over/under values for nodes of 7.35%/18.12% and zones of 0.24%/0.22%, still pretty good considering the crazy weight ranges.

Summary

Hopefully this series has been a good introduction to building a ring. This code is essentially how the OpenStack Swift ring works, except that Swifts ring has lots of additional optimizations, such as storing each replica assignment separately, and lots of extra features for building, validating, and otherwise working with rings.

2.21 Modifying Ring Partition Power

The ring partition power determines the on-disk location of data files and is selected when creating a new ring. In normal operation, it is a fixed value. This is because a different partition power results in a different on-disk location for all data files.

However, increasing the partition power by 1 can be done by choosing locations that are on the same disk. As a result, we can create hard-links for both the new and old locations, avoiding data movement without impacting availability.

To enable a partition power change without interrupting user access, object servers need to be aware of it in advance. Therefore a partition power change needs to be done in multiple steps.

Note: Do not increase the partition power on account and container rings. Increasing the partition power is *only* supported for object rings. Trying to increase the part_power for account and container rings *will* result in unavailability, maybe even data loss.

2.21.1 Caveats

Before increasing the partition power, consider the possible drawbacks. There are a few caveats when increasing the partition power:

- Almost all diskfiles in the cluster need to be relinked then cleaned up, and all partition directories need to be rehashed. This imposes significant I/O load on object servers, which may impact client requests. Consider using cgroups, `ionice`, or even just the built-in `--files-per-second` rate-limiting to reduce client impact.
- Object replicators and reconstructors will skip affected policies during the partition power increase. Replicators are not aware of hard-links, and would simply copy the content; this would result in heavy data movement and the worst case would be that all data is stored twice.
- Due to the fact that each object will now be hard linked from two locations, many more inodes will be used temporarily - expect around twice the amount. You need to check the free inode count *before* increasing the partition power. Even after the increase is complete and extra hardlinks are cleaned up, expect increased inode usage since there will be twice as many partition and suffix directories.
- Also, object auditors might read each object twice before cleanup removes the second hard link.
- Due to the new inodes more memory is needed to cache them, and your object servers should have plenty of available memory to avoid running out of inode cache. Setting `vfs_cache_pressure` to 1 might help with that.
- All nodes in the cluster *must* run at least Swift version 2.13.0 or later.

Due to these caveats you should only increase the partition power if really needed, i.e. if the number of partitions per disk is extremely low and the data is distributed unevenly across disks.

2.21.2 1. Prepare partition power increase

The swift-ring-builder is used to prepare the ring for an upcoming partition power increase. It will store a new variable `next_part_power` with the current partition power + 1. Object servers recognize this, and hard links to the new location will be created (or deleted) on every PUT or DELETE. This will make it possible to access newly written objects using the future partition power:

```
swift-ring-builder <builder-file> prepare_increase_partition_power
swift-ring-builder <builder-file> write_ring
```

Now you need to copy the updated .ring.gz to all nodes. Already existing data needs to be relinked too; therefore an operator has to run a relinker command on all object servers in this phase:

```
swift-object-relinker relink
```

Note: Start relinking after *all* the servers re-read the modified ring files, which normally happens within 15 seconds after writing a modified ring. Also, make sure the modified rings are pushed to all nodes running object services (replicators, reconstructors and reconcilers)- they have to skip the policy during relinking.

Note: The relinking command must run as the same user as the daemon processes (usually swift). It will

create files and directories that must be manipulable by the daemon processes (server, auditor, replicator,). If necessary, the `--user` option may be used to drop privileges.

Relinking might take some time; while there is no data copied or actually moved, the tool still needs to walk the whole file system and create new hard links as required.

2.21.3 2. Increase partition power

Now that all existing data can be found using the new location, its time to actually increase the partition power itself:

```
swift-ring-builder <builder-file> increase_partition_power
swift-ring-builder <builder-file> write_ring
```

Now you need to copy the updated .ring.gz again to all nodes. Object servers are now using the new, increased partition power and no longer create additional hard links.

Note: The object servers will create additional hard links for each modified or new object, and this requires more inodes.

Note: If you decide you dont want to increase the partition power, you should instead cancel the increase. It is not possible to revert this operation once started. To abort the partition power increase, execute the following commands, copy the updated .ring.gz files to all nodes and continue with *3. Cleanup* afterwards:

```
swift-ring-builder <builder-file> cancel_increase_partition_power
swift-ring-builder <builder-file> write_ring
```

2.21.4 3. Cleanup

Existing hard links in the old locations need to be removed, and a cleanup tool is provided to do this. Run the following command on each storage node:

```
swift-object-relinker cleanup
```

Note: The cleanup must be finished within your object servers `reclaim_age` period (which is by default 1 week). Otherwise objects that have been overwritten between step #1 and step #2 and deleted afterwards cant be cleaned up anymore. You may want to increase your `reclaim_age` before or during relinking.

Afterwards it is required to update the rings one last time to inform servers that all steps to increase the partition power are done, and replicators should resume their job:

```
swift-ring-builder <builder-file> finish_increase_partition_power
swift-ring-builder <builder-file> write_ring
```

Now you need to copy the updated .ring.gz again to all nodes.

2.21.5 Background

An existing object that is currently located on partition X will be placed either on partition 2*X or 2*X+1 after the partition power is increased. The reason for this is the Ring.get_part() method, that does a bitwise shift to the right.

To avoid actual data movement to different disks or even nodes, the allocation of partitions to nodes needs to be changed. The allocation is pairwise due to the above mentioned new partition scheme. Therefore devices are allocated like this, with the partition being the index and the value being the device id:

```
    old         new
part  dev   part  dev
----  ---   ----  ---
0     0     0     0
                  1     0
1     3     2     3
                  3     3
2     7     4     7
                  5     7
3     5     6     5
                  7     5
4     2     8     2
                  9     2
5     1     10    1
                  11    1
```

There is a helper method to compute the new path, and the following example shows the mapping between old and new location:

```
>>> from swift.common.utils import replace_partition_in_path
>>> old='objects/16003/a38/fa0fcec07328d068e24ccbf2a62f2a38/1467658208.57179.
↪data'
>>> replace_partition_in_path('', '/sda/' + old, 14)
'objects/16003/a38/fa0fcec07328d068e24ccbf2a62f2a38/1467658208.57179.data'
>>> replace_partition_in_path('', '/sda/' + old, 15)
'objects/32007/a38/fa0fcec07328d068e24ccbf2a62f2a38/1467658208.57179.data'
```

Using the original partition power (14) it returned the same path; however after an increase to 15 it returns the new path, and the new partition is 2*X+1 in this case.

2.22 Associated Projects

2.22.1 Application Bindings

- OpenStack supported binding:
  - Python-SwiftClient
- Unofficial libraries and bindings:
  - PHP

    * PHP-opencloud - Official Rackspace PHP bindings that should work for other Swift deployments too.

  - Ruby
    * swift_client - Small but powerful Ruby client to interact with OpenStack Swift
    * nightcrawler_swift - This Ruby gem teleports your assets to an OpenStack Swift bucket/container
    * swift storage - Simple OpenStack Swift storage client.
  - Java
    * libcloud - Apache Libcloud - a unified interface in Python for different clouds with OpenStack Swift support.
    * jclouds - Java library offering bindings for all OpenStack projects
    * java-openstack-swift - Java bindings for OpenStack Swift
    * javaswift - Collection of Java tools for Swift
  - Bash
    * supload - Bash script to upload file to cloud storage based on OpenStack Swift API.
  - .NET
    * openstacknetsdk.org - An OpenStack Cloud SDK for Microsoft .NET.
  - Go
    * Go language bindings
    * Gophercloud an OpenStack SDK for Go

2.22.2 Authentication

- Keystone - Official Identity Service for OpenStack.
- Swauth - **RETIRED**: An alternative Swift authentication service that only requires Swift itself.
- Basicauth - HTTP Basic authentication support (keystone backed).

2.22.3 Command Line Access

- Swiftly - Alternate command line access to Swift with direct (no proxy) access capabilities as well.

2.22.4 Log Processing

- slogging - Basic stats and logging tools.

2.22.5 Monitoring & Statistics

- Swift Informant - Swift proxy Middleware to send events to a statsd instance.
- Swift Inspector - Swift middleware to relay information about a request back to the client.

2.22.6 Content Distribution Network Integration

- SOS - Swift Origin Server.

2.22.7 Alternative API

- ProxyFS - Integrated file and object access for Swift object storage
- SwiftHLM - a middleware for using OpenStack Swift with tape and other high latency media storage backends.

2.22.8 Benchmarking/Load Generators

- getput - getput tool suite
- COSbench - COSbench tool suite

2.22.9 Custom Logger Hooks

- swift-sentry - Sentry exception reporting for Swift

2.22.10 Storage Backends (DiskFile API implementations)

- Swift-on-File - Enables objects created using Swift API to be accessed as files on a POSIX filesystem and vice versa.
- swift-scality-backend - Scality sproxyd object server implementation for Swift.

2.22.11 Developer Tools

- SAIO bash scripts - Well commented simple bash scripts for Swift all in one setup.
- vagrant-swift-all-in-one - Quickly setup a standard development environment using Vagrant and Chef cookbooks in an Ubuntu virtual machine.
- SAIO Ansible playbook - Quickly setup a standard development environment using Vagrant and Ansible in a Fedora virtual machine (with built-in Swift-on-File support).
- Multi Swift - Bash scripts to spin up multiple Swift clusters sharing the same hardware

2.22.12 Other

- Glance - Provides services for discovering, registering, and retrieving virtual machine images (for OpenStack Compute [Nova], for example).
- Django Swiftbrowser - Simple Django web app to access OpenStack Swift.
- Swift-account-stats - Swift-account-stats is a tool to report statistics on Swift usage at tenant and global levels.
- PyECLib - High-level erasure code library used by Swift
- liberasurecode - Low-level erasure code library used by PyECLib
- Swift Browser - JavaScript interface for Swift
- swift-ui - OpenStack Swift web browser
- swiftbackmeup - Utility that allows one to create backups and upload them to OpenStack Swift

CHAPTER

THREE

CONTRIBUTOR DOCUMENTATION

3.1 Contributing to OpenStack Swift

3.1.1 Who is a Contributor?

Put simply, if you improve Swift, youre a contributor. The easiest way to improve the project is to tell us where theres a bug. In other words, filing a bug is a valuable and helpful way to contribute to the project.

Once a bug has been filed, someone will work on writing a patch to fix the bug. Perhaps youd like to fix a bug. Writing code to fix a bug or add new functionality is tremendously important.

Once code has been written, it is submitted upstream for review. All code, even that written by the most senior members of the community, must pass code review and all tests before it can be included in the project. Reviewing proposed patches is a very helpful way to be a contributor.

Swift is nothing without the community behind it. Wed love to welcome you to our community. Come find us in #openstack-swift on OFTC IRC or on the OpenStack dev mailing list.

For general information on contributing to OpenStack, please check out the contributor guide to get started. It covers all the basics that are common to all OpenStack projects: the accounts you need, the basics of interacting with our Gerrit review system, how we communicate as a community, etc.

If you want more Swift related project documentation make sure you checkout the Swift developer (contributor) documentation at https://docs.openstack.org/swift/latest/

Filing a Bug

Filing a bug is the easiest way to contribute. We use Launchpad as a bug tracker; you can find currently-tracked bugs at https://bugs.launchpad.net/swift. Use the Report a bug link to file a new bug.

If you find something in Swift that doesnt match the documentation or doesnt meet your expectations with how it should work, please let us know. Of course, if you ever get an error (like a Traceback message in the logs), we definitely want to know about that. Well do our best to diagnose any problem and patch it as soon as possible.

A bug report, at minimum, should describe what you were doing that caused the bug. Swift broke, pls fix is not helpful. Instead, something like When I restarted syslog, Swift started logging traceback messages is very helpful. The goal is that we can reproduce the bug and isolate the issue in order to apply a fix. If you dont have full details, thats ok. Anything you can provide is helpful.

You may have noticed that there are many tracked bugs, but not all of them have been confirmed. If you take a look at an old bug report and you can reproduce the issue described, please leave a comment on the bug about that. It lets us all know that the bug is very likely to be valid.

Reviewing Someone Elses Code

All code reviews in OpenStack projects are done on https://review.opendev.org/. Reviewing patches is one of the most effective ways you can contribute to the community.

Weve written REVIEW_GUIDELINES.rst (found in this source tree) to help you give good reviews.

https://wiki.openstack.org/wiki/Swift/PriorityReviews is a starting point to find what reviews are priority in the community.

3.1.2 What do I work on?

If youre looking for a way to write and contribute code, but youre not sure what to work on, check out the wishlist bugs in the bug tracker. These are normally smaller items that someone took the time to write down but didnt have time to implement.

And please join #openstack-swift on OFTC IRC to tell us what youre working on.

3.1.3 Getting Started

https://docs.openstack.org/swift/latest/first_contribution_swift.html

Once those steps have been completed, changes to OpenStack should be submitted for review via the Gerrit tool, following the workflow documented at http://docs.openstack.org/infra/manual/developers.html#development-workflow.

Gerrit is the review system used in the OpenStack projects. Were sorry, but we wont be able to respond to pull requests submitted through GitHub.

Bugs should be filed on Launchpad, not in GitHubs issue tracker.

3.2 Swift Design Principles

- The Zen of Python
- Simple Scales
- Minimal dependencies
- Re-use existing tools and libraries when reasonable
- Leverage the economies of scale
- Small, loosely coupled RESTful services
- No single points of failure
- Start with the use case
- then design from the cluster operator up
- If you havent argued about it, you dont have the right answer yet :)
- If it is your first implementation, you probably arent done yet :)

Please dont feel offended by difference of opinion. Be prepared to advocate for your change and iterate on it based on feedback. Reach out to other people working on the project on IRC or the mailing list - we want to help.

3.3 Recommended workflow

- Set up a Swift All-In-One VM(SAIO).
- Make your changes. Docs and tests for your patch must land before or with your patch.
- Run unit tests, functional tests, probe tests `./.unittests ./.functests ./.probetests`
- Run `tox` (no command-line args needed)
- `git review`

3.4 Notes on Testing

Running the tests above against Swift in your development environment (ie your SAIO) will catch most issues. Any patch you propose is expected to be both tested and documented and all tests should pass.

If you want to run just a subset of the tests while you are developing, you can use nosetests:

```
cd test/unit/common/middleware/ && nosetests test_healthcheck.py
```

To check which parts of your code are being exercised by a test, you can run tox and then point your browser to swift/cover/index.html:

```
tox -e py27 -- test.unit.common.middleware.test_healthcheck:TestHealthCheck.
↪test_healthcheck
```

Swifts unit tests are designed to test small parts of the code in isolation. The functional tests validate that the entire system is working from an external perspective (they are black-box tests). You can even run functional tests against public Swift endpoints. The probetests are designed to test much of Swifts internal processes. For example, a test may write data, intentionally corrupt it, and then ensure that the correct processes detect and repair it.

When your patch is submitted for code review, it will automatically be tested on the OpenStack CI infrastructure. In addition to many of the tests above, it will also be tested by several other OpenStack test jobs.

Once your patch has been reviewed and approved by core reviewers and has passed all automated tests, it will be merged into the Swift source tree.

3.5 Ideas

https://wiki.openstack.org/wiki/Swift/ideas

If youre working on something, its a very good idea to write down what youre thinking about. This lets others get up to speed, helps you collaborate, and serves as a great record for future reference. Write down your thoughts somewhere and put a link to it here. It doesnt matter what form your thoughts are in; use whatever is best for you. Your document should include why your idea is needed and your thoughts on particular design choices and tradeoffs. Please include some contact information (ideally, your IRC nick) so that people can collaborate with you.

3.6 Community

3.6.1 Communication

IRC People working on the Swift project may be found in the `#openstack-swift` channel on OFTC during working hours in their timezone. The channel is logged, so if you ask a question when no one is around, you can check the log to see if its been answered: http://eavesdrop.openstack.org/irclogs/%23openstack-swift/

weekly meeting This is a Swift team meeting. The discussion in this meeting is about all things related to the Swift project:

- time: http://eavesdrop.openstack.org/#Swift_Team_Meeting
- agenda: https://wiki.openstack.org/wiki/Meetings/Swift

mailing list We use the openstack-discuss@lists.openstack.org mailing list for asynchronous discussions or to communicate with other OpenStack teams. Use the prefix `[swift]` in your subject line (its a high-volume list, so most people use email filters).

More information about the mailing list, including how to subscribe and read the archives, can be found at: http://lists.openstack.org/cgi-bin/mailman/listinfo/openstack-discuss

3.6.2 Contacting the Core Team

The swift-core team is an active group of contributors who are responsible for directing and maintaining the Swift project. As a new contributor, your interaction with this group will be mostly through code reviews, because only members of swift-core can approve a code change to be merged into the code repository. But the swift-core team also spend time on IRC so feel free to drop in to ask questions or just to meet us.

Note: Although your contribution will require reviews by members of swift-core, these arent the only people whose reviews matter. Anyone with a gerrit account can post reviews, so you can ask other developers you know to review your code and you can review theirs. (A good way to learn your way around the codebase is to review other peoples patches.)

If youre thinking, Im new at this, how can I possibly provide a helpful review?, take a look at How to Review Changes the OpenStack Way.

Or for more specifically in a Swift context read *Review Guidelines*

You can learn more about the role of core reviewers in the OpenStack governance documentation: https://docs.openstack.org/contributors/common/governance.html#core-reviewer

The membership list of swift-core is maintained in gerrit: https://review.opendev.org/#/admin/groups/24,members

You can also find the members of the swift-core team at the Swift weekly meetings.

3.6.3 Getting Your Patch Merged

Understanding how reviewers review and what they look for will help getting your code merged. See Swift Review Guidelines for how we review code.

Keep in mind that reviewers are also human; if something feels stalled, then come and poke us on IRC or add it to our meeting agenda.

3.6.4 Project Team Lead Duties

All common PTL duties are enumerated in the PTL guide.

3.7 Review Guidelines

Effective code review is a skill like any other professional skill you develop with experience. Effective code review requires trust. No one is perfect. Everyone makes mistakes. Trust builds over time.

This document will enumerate behaviors commonly observed and associated with competent reviews of changes purposed to the Swift code base. No one is expected to follow these steps. Guidelines are not *rules*, not all behaviors will be relevant in all situations.

> Code review is collaboration, not judgement.
>
> —Alistair Coles

3.7.1 Checkout the Change

You will need to have a copy of the change in an environment where you can freely edit and experiment with the code in order to provide a non-superficial review. Superficial reviews are not terribly helpful. Always try to be helpful. ;)

Check out the change so that you may begin.

Commonly, `git review -d <change-id>`

3.7.2 Run it

Imagine that you submit a patch to Swift, and a reviewer starts to take a look at it. Your commit message on the patch claims that it fixes a bug or adds a feature, but as soon as the reviewer downloads it locally and tries to test it, a severe and obvious error shows up. Something like a syntax error or a missing dependency.

Did you even run this? is the review comment all contributors dread.

Reviewers in particular need to be fearful merging changes that just dont work - or at least fail in frequently common enough scenarios to be considered horribly broken. A comment in our review that says roughly I ran this on my machine and observed `description of behavior change is supposed to achieve` is the most powerful defense we have against the terrible scorn from our fellow Swift developers and operators when we accidentally merge bad code.

If youre doing a fair amount of reviews - you will participate in merging a change that will break my clusters - its cool - Ill do it to you at some point too (sorry about that). But when either of us go look at the reviews to understand the process gap that allowed this to happen - it better not be just because we were too lazy to check it out and run it before it got merged.

Or be warned, you may receive, the dreaded

> Did you even *run* this?

Im sorry, I know its rough. ;)

3.7.3 Consider edge cases very seriously

> Saying that should rarely happen is the same as saying that *will* happen
>
> —Douglas Crockford

Scale is an *amazingly* abusive partner. If you contribute changes to Swift your code is running - in production - at scale - and your bugs cannot hide. I wish on all of us that our bugs may be exceptionally rare - meaning they only happen in extremely unlikely edge cases. For example, bad things that happen only 1 out of every 10K times an op is performed will be discovered in minutes. Bad things that happen only 1 out of every one billion times something happens will be observed - by multiple deployments - over the course of a release. Bad things that happen 1/100 times some op is performed are considered horribly broken. Tests must exhaustively exercise possible scenarios. Every system call and network connection will raise an error and timeout - where will that Exception be caught?

3.7.4 Run the tests

Yes, I know Gerrit does this already. You can do it *too*. You might not need to re-run *all* the tests on your machine - it depends on the change. But, if youre not sure which will be most useful - running all of them best - unit - functional - probe. If you cant reliably get all tests passing in your development environment you will not be able to do effective reviews. Whatever tests/suites you are able to exercise/validate on your machine against your config you should mention in your review comments so that other reviewers might choose to do *other* testing locally when they have the change checked out.

e.g.

> I went ahead and ran probe/test_object_metadata_replication.py on my machine with both sync_method = rsync and sync_method = ssync - that works for me - but I didnt try it with object_post_as_copy = false

3.7.5 Maintainable Code is Obvious

Style is an important component to review. The goal is maintainability.

However, keep in mind that generally style, readability and maintainability are orthogonal to the suitability of a change for merge. A critical bug fix may be a well written pythonic masterpiece of style - or it may be a hack-y ugly mess that will absolutely need to be cleaned up at some point - but it absolutely should merge because: CRITICAL. BUG. FIX.

You should comment inline to praise code that is obvious. You should comment inline to highlight code that you found to be obfuscated.

Unfortunately readability is often subjective. We should remember that its probably just our own personal preference. Rather than a comment that says You should use a list comprehension here - rewrite the code as a list comprehension, run the specific tests that hit the relevant section to validate your code is correct, then leave a comment that says:

> I find this more readable:
>
> `diff with working tested code`

If the author (or another reviewer) agrees - its possible the change will get updated to include that improvement before it is merged; or it may happen in a follow-up change.

However, remember that style is non-material - it is useful to provide (via diff) suggestions to improve maintainability as part of your review - but if the suggestion is functionally equivalent - it is by definition optional.

3.7.6 Commit Messages

Read the commit message thoroughly before you begin the review.

Commit messages must answer the why and the what for - more so than the how or what it does. Commonly this will take the form of a short description:

- What is broken - without this change
- What is impossible to do with Swift - without this change
- What is slower/worse/harder - without this change

If youre not able to discern why a change is being made or how it would be used - you may have to ask for more details before you can successfully review it.

Commit messages need to have a high consistent quality. While many things under source control can be fixed and improved in a follow-up change - commit messages are forever. Luckily its easy to fix minor mistakes using the in-line edit feature in Gerrit! If you can avoid ever having to *ask* someone to change a commit message you will find yourself an amazingly happier and more productive reviewer.

Also commit messages should follow the OpenStack Commit Message guidelines, including references to relevant impact tags or bug numbers. You should hand out links to the OpenStack Commit Message guidelines *liberally* via comments when fixing commit messages during review.

Here you go: GitCommitMessages

3.7.7 New Tests

New tests should be added for all code changes. Historically you should expect good changes to have a diff line count ratio of at least 2:1 tests to code. Even if a change has to fix a lot of *existing* tests, if a change does not include any *new* tests it probably should not merge.

If a change includes a good ratio of test changes and adds new tests - you should say so in your review comments.

If it does not - you should write some!

and offer them to the patch author as a diff indicating to them that something like these tests Im providing as an example will *need* to be included in this change before it is suitable to merge. Bonus points if you include suggestions for the author as to how they might improve or expand upon the tests stubs you provide.

Be *very* careful about asking an author to add a test for a small change before attempting to do so yourself. Its quite possible there is a lack of existing test infrastructure needed to develop a concise and clear test - the author of a small change may not be the best person to introduce a large amount of new test infrastructure. Also, most of the time remember its *harder* to write the test than the change - if the author is unable to develop a test for their change on their own you may prevent a useful change from being merged. At a minimum you should suggest a specific unit test that you think they should be able to copy and modify to exercise the behavior in their change. If youre not sure if such a test exists - replace their change with an Exception and run tests until you find one that blows up.

3.7.8 Documentation

Most changes should include documentation. New functions and code should have Docstrings. Tests should obviate new or changed behaviors with descriptive and meaningful phrases. New features should include changes to the documentation tree. New config options should be documented in example configs. The commit message should document the change for the change log.

Always point out typos or grammar mistakes when you see them in review, but also consider that if you were able to recognize the intent of the statement - documentation with typos may be easier to iterate and improve on than nothing.

If a change does not have adequate documentation it may not be suitable to merge. If a change includes incorrect or misleading documentation or is contrary to *existing* documentation is probably is not suitable to merge.

Every change could have better documentation.

Like with tests, a patch isnt done until it has docs. Any patch that adds a new feature, changes behavior, updates configs, or in any other way is different than previous behavior requires docs. manpages, sample configs, docstrings, descriptive prose in the source tree, etc.

3.7.9 Reviewers Write Code

Reviews have been shown to provide many benefits - one of which is shared ownership. After providing a positive review you should understand how the change works. Doing this will probably require you to play with the change.

You might functionally test the change in various scenarios. You may need to write a new unit test to validate the change will degrade gracefully under failure. You might have to write a script to exercise the change under some superficial load. You might have to break the change and validate the new tests fail and provide useful errors. You might have to step through some critical section of the code in a debugger to understand when all the possible branches are exercised in tests.

When youre done with your review an artifact of your effort will be observable in the piles of code and scripts and diffs you wrote while reviewing. You should make sure to capture those artifacts in a paste or gist and include them in your review comments so that others may reference them.

e.g.

> When I broke the change like this:
>
> ```
> diff
> ```
>
> it blew up like this:
>
> ```
> unit test failure
> ```

Its not uncommon that a review takes more time than writing a change - hopefully the author also spent as much time as you did *validating* their change but thats not really in your control. When you provide a positive review you should be sure you understand the change - even seemingly trivial changes will take time to consider the ramifications.

3.7.10 Leave Comments

Leave. Lots. Of. Comments.

A popular web comic has stated that WTFs/Minute is the *only* valid measurement of code quality.

If something initially strikes you as questionable - you should jot down a note so you can loop back around to it.

However, because of the distributed nature of authors and reviewers its *imperative* that you try your best to answer your own questions as part of your review.

Do not say Does this blow up if it gets called when xyz - rather try and find a test that specifically covers that condition and mention it in the comment so others can find it more quickly. Or if you can find no such test, add one to demonstrate the failure, and include a diff in a comment. Hopefully you can say I *thought* this would blow up, so I wrote this test, but it seems fine.

But if your initial reaction is I dont understand this or How does this even work? you should notate it and explain whatever you *were* able to figure out in order to help subsequent reviewers more quickly identify and grok the subtle or complex issues.

Because you will be leaving lots of comments - many of which are potentially not highlighting anything specific - it is VERY important to leave a good summary. Your summary should include details of how you reviewed the change. You may include what you liked most, or least.

If you are leaving a negative score ideally you should provide clear instructions on how the change could be modified such that it would be suitable for merge - again diffs work best.

3.7.11 Scoring

Scoring is subjective. Try to realize youre making a judgment call.

A positive score means you believe Swift would be undeniably better off with this code merged than it would be going one more second without this change running in production immediately. It is indeed high praise - you should be sure.

A negative score means that to the best of your abilities you have not been able to your satisfaction, to justify the value of a change against the cost of its deficiencies and risks. It is a surprisingly difficult chore to be confident about the value of unproven code or a not well understood use-case in an uncertain world, and unfortunately all too easy with a **thorough** review to uncover our defects, and be reminded of the risk of regression.

Reviewers must try *very* hard first and foremost to keep master stable.

If you can demonstrate a change has an incorrect *behavior* its almost without exception that the change must be revised to fix the defect *before* merging rather than letting it in and having to also file a bug.

Every commit must be deployable to production.

Beyond that - almost any change might be merge-able depending on its merits! Here are some tips you might be able to use to find more changes that should merge!

1. Fixing bugs is HUGELY valuable - the *only* thing which has a higher cost than the value of fixing a bug - is adding a new bug - if its broken and this change makes it fixed (without breaking anything else) you have a winner!
2. Features are INCREDIBLY difficult to justify their value against the cost of increased complexity, lowered maintainability, risk of regression, or new defects. Try to focus on what is *impossible* without the feature - when you make the impossible possible, things are better. Make things better.
3. Purely test/doc changes, complex refactoring, or mechanical cleanups are quite nuanced because theres less concrete objective value. Ive seen lots of these kind of changes get lost to the backlog. Ive also seen some success where multiple authors have collaborated to push-over a change rather than provide a review ultimately resulting in a quorum of three or more authors who all agree there is a lot of value in the change - however subjective.

Because the bar is high - most reviews will end with a negative score.

However, for non-material grievances (nits) - you should feel confident in a positive review if the change is otherwise complete correct and undeniably makes Swift better (not perfect, *better*). If you see something worth fixing you should point it out in review comments, but when applying a score consider if it *need* be fixed before the change is suitable to merge vs. fixing it in a follow up change? Consider if the change makes Swift so undeniably *better* and it was deployed in production without making any additional changes would it still be correct and complete? Would releasing the change to production without any additional follow up make it more difficult to maintain and continue to improve Swift?

Endeavor to leave a positive or negative score on every change you review.

Use your best judgment.

3.7.12 A note on Swift Core Maintainers

Swift Core maintainers may provide positive reviews scores that *look* different from your reviews - a +2 instead of a +1.

But its *exactly the same* as your +1.

It means the change has been thoroughly and positively reviewed. The only reason its different is to help identify changes which have received multiple competent and positive reviews. If you consistently provide competent reviews you run a *VERY* high risk of being approached to have your future positive review scores changed from a +1 to +2 in order to make it easier to identify changes which need to get merged.

Ideally a review from a core maintainer should provide a clear path forward for the patch author. If you dont know how to proceed respond to the reviewers comments on the change and ask for help. Wed love to try and help.

CHAPTER

FOUR

DEVELOPER DOCUMENTATION

4.1 Development Guidelines

4.1.1 Coding Guidelines

For the most part we try to follow PEP 8 guidelines which can be viewed here: http://www.python.org/dev/peps/pep-0008/

4.1.2 Testing Guidelines

Swift has a comprehensive suite of tests and pep8 checks that are run on all submitted code, and it is recommended that developers execute the tests themselves to catch regressions early. Developers are also expected to keep the test suite up-to-date with any submitted code changes.

Swifts tests and pep8 checks can be executed in an isolated environment with `tox`: http://tox.testrun.org/

To execute the tests:

- Ensure `pip` and `virtualenv` are upgraded to satisfy the version requirements listed in the OpenStack global requirements:

```
pip install pip -U
pip install virtualenv -U
```

- Install `tox`:

```
pip install tox
```

- Generate list of distribution packages to install for testing:

```
tox -e bindep
```

  Now install these packages using your distribution package manager like apt-get, dnf, yum, or zypper.

- Run `tox` from the root of the swift repo:

```
tox
```

Note: If you installed using `cd ~/swift; sudo python setup.py develop`, you may need to do `cd ~/swift; sudo chown -R ${USER}:${USER} swift.egg-info` prior to running `tox`.

- By default `tox` will run all of the unit test and pep8 checks listed in the `tox.ini` file `envlist` option. A subset of the test environments can be specified on the `tox` command line or by setting the `TOXENV` environment variable. For example, to run only the pep8 checks and python2.7 unit tests use:

```
tox -e pep8,py27
```

or:

```
TOXENV=py27,pep8 tox
```

Note: As of `tox` version 2.0.0, most environment variables are not automatically passed to the test environment. Swifts `tox.ini` overrides this default behavior so that variable names matching `SWIFT_*` and `*_proxy` will be passed, but you may need to run `tox --recreate` for this to take effect after upgrading from `tox` <2.0.0.

Conversely, if you do not want those environment variables to be passed to the test environment then you will need to unset them before calling `tox`.

Also, if you ever encounter DistributionNotFound, try to use `tox --recreate` or remove the `.tox` directory to force `tox` to recreate the dependency list.

Swifts tests require having an XFS directory available in `/tmp` or in the `TMPDIR` environment variable.

Swifts functional tests may be executed against a *SAIO (Swift All In One)* or other running Swift cluster using the command:

```
tox -e func
```

The endpoint and authorization credentials to be used by functional tests should be configured in the `test.conf` file as described in the section *Setting up scripts for running Swift*.

The environment variable `SWIFT_TEST_POLICY` may be set to specify a particular storage policy *name* that will be used for testing. When set, tests that would otherwise not specify a policy or choose a random policy from those available will instead use the policy specified. Tests that use more than one policy will include the specified policy in the set of policies used. The specified policy must be available on the cluster under test.

For example, this command would run the functional tests using policy silver:

```
SWIFT_TEST_POLICY=silver tox -e func
```

To run a single functional test, use the `--no-discover` option together with a path to a specific test method, for example:

```
tox -e func -- --no-discover test.functional.tests.TestFile.testCopy
```

In-process functional testing

If the `test.conf` file is not found then the functional test framework will instantiate a set of Swift servers in the same process that executes the functional tests. This in-process test mode may also be enabled (or disabled) by setting the environment variable `SWIFT_TEST_IN_PROCESS` to a true (or false) value prior to executing `tox -e func`.

When using the in-process test mode some server configuration options may be set using environment variables:

- the optional in-memory object server may be selected by setting the environment variable `SWIFT_TEST_IN_MEMORY_OBJ` to a true value.
- encryption may be added to the proxy pipeline by setting the environment variable `SWIFT_TEST_IN_PROCESS_CONF_LOADER` to `encryption`.
- a 2+1 EC policy may be installed as the default policy by setting the environment variable `SWIFT_TEST_IN_PROCESS_CONF_LOADER` to `ec`.
- logging to stdout may be enabled by setting `SWIFT_TEST_DEBUG_LOGS`.

For example, this command would run the in-process mode functional tests with encryption enabled in the proxy-server:

```
SWIFT_TEST_IN_PROCESS=1 SWIFT_TEST_IN_PROCESS_CONF_LOADER=encryption \
    tox -e func
```

This particular example may also be run using the `func-encryption` tox environment:

```
tox -e func-encryption
```

The `tox.ini` file also specifies test environments for running other in-process functional test configurations, e.g.:

```
tox -e func-ec
```

To debug the functional tests, use the in-process test mode and pass the `--pdb` flag to `tox`:

```
SWIFT_TEST_IN_PROCESS=1 tox -e func -- --pdb \
    test.functional.tests.TestFile.testCopy
```

The in-process test mode searches for `proxy-server.conf` and `swift.conf` config files from which it copies config options and overrides some options to suit in process testing. The search will first look for config files in a `<custom_conf_source_dir>` that may optionally be specified using the environment variable:

```
SWIFT_TEST_IN_PROCESS_CONF_DIR=<custom_conf_source_dir>
```

If `SWIFT_TEST_IN_PROCESS_CONF_DIR` is not set, or if a config file is not found in `<custom_conf_source_dir>`, the search will then look in the `etc/` directory in the source tree. If the config file is still not found, the corresponding sample config file from `etc/` is used (e.g. `proxy-server.conf-sample` or `swift.conf-sample`).

When using the in-process test mode `SWIFT_TEST_POLICY` may be set to specify a particular storage policy *name* that will be used for testing as described above. When set, this policy must exist in the

swift.conf file and its corresponding ring file must exist in <custom_conf_source_dir> (if specified) or etc/. The test setup will set the specified policy to be the default and use its ring file properties for constructing the test object ring. This allows in-process testing to be run against various policy types and ring files.

For example, this command would run the in-process mode functional tests using config files found in $HOME/my_tests and policy silver:

```
SWIFT_TEST_IN_PROCESS=1 SWIFT_TEST_IN_PROCESS_CONF_DIR=$HOME/my_tests \
   SWIFT_TEST_POLICY=silver tox -e func
```

S3 API cross-compatibility tests

The cross-compatibility tests in directory *test/s3api* are intended to verify that the Swift S3 API behaves in the same way as the AWS S3 API. They should pass when run against either a Swift endpoint (with S3 API enabled) or an AWS S3 endpoint.

To run against an AWS S3 endpoint, the */etc/swift/test.conf* file must be edited to provide AWS key IDs and secrets. Alternatively, an AWS CLI style credentials file can be loaded by setting the SWIFT_TEST_AWS_CONFIG_FILE environment variable, e.g.:

```
SWIFT_TEST_AWS_CONFIG_FILE=~/.aws/credentials nosetests ./test/s3api
```

Note: When using SWIFT_TEST_AWS_CONFIG_FILE, the region defaults to us-east-1 and only the default credentials are loaded.

4.1.3 Coding Style

Swift uses flake8 with the OpenStack hacking module to enforce coding style.

Install flake8 and hacking with pip or by the packages of your Operating System.

It is advised to integrate flake8+hacking with your editor to get it automated and not get *caught* by Jenkins.

For example for Vim the syntastic plugin can do this for you.

4.1.4 Documentation Guidelines

The documentation in docstrings should follow the PEP 257 conventions (as mentioned in the PEP 8 guidelines).

More specifically:

1. Triple quotes should be used for all docstrings.
2. If the docstring is simple and fits on one line, then just use one line.
3. For docstrings that take multiple lines, there should be a newline after the opening quotes, and before the closing quotes.

4. Sphinx is used to build documentation, so use the restructured text markup to designate parameters, return values, etc. Documentation on the sphinx specific markup can be found here: http://sphinx.pocoo.org/markup/index.html

To build documentation run:

```
pip install -r requirements.txt -r doc/requirements.txt
sphinx-build -W -b html doc/source doc/build/html
```

and then browse to doc/build/html/index.html. These docs are auto-generated after every commit and available online at https://docs.openstack.org/swift/latest/.

4.1.5 Manpages

For sanity check of your change in manpage, use this command in the root of your Swift repo:

```
./.manpages
```

4.1.6 License and Copyright

You can have the following copyright and license statement at the top of each source file. Copyright assignment is optional.

New files should contain the current year. Substantial updates can have another year added, and date ranges are not needed.:

```
# Copyright (c) 2013 OpenStack Foundation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

4.2 SAIO (Swift All In One)

Note: This guide assumes an existing Linux server. A physical machine or VM will work. We recommend configuring it with at least 2GB of memory and 40GB of storage space. We recommend using a VM in order to isolate Swift and its dependencies from other projects you may be working on.

4.2.1 Instructions for setting up a development VM

This section documents setting up a virtual machine for doing Swift development. The virtual machine will emulate running a four node Swift cluster. To begin:

- Get a Linux system server image, this guide will cover:
  - Ubuntu 14.04, 16.04 LTS
  - CentOS 7
  - Fedora
  - OpenSuse
- Create guest virtual machine from the image.

4.2.2 Whats in a <your-user-name>

Much of the configuration described in this guide requires escalated administrator (`root`) privileges; however, we assume that administrator logs in as an unprivileged user and can use `sudo` to run privileged commands.

Swift processes also run under a separate user and group, set by configuration option, and referenced as `<your-user-name>:<your-group-name>`. The default user is `swift`, which may not exist on your system. These instructions are intended to allow a developer to use his/her username for `<your-user-name>:<your-group-name>`.

Note: For OpenSuse users, a users primary group is `users`, so you have 2 options:

- Change `${USER}:${USER}` to `${USER}:users` in all references of this guide; or
- Create a group for your username and add yourself to it:

```
sudo groupadd ${USER} && sudo gpasswd -a ${USER} ${USER} && newgrp ${USER}
```

4.2.3 Installing dependencies

- On `apt` based systems:

```
sudo apt-get update
sudo apt-get install curl gcc memcached rsync sqlite3 xfsprogs \
                     git-core libffi-dev python-setuptools \
                     liberasurecode-dev libssl-dev
sudo apt-get install python-coverage python-dev python-nose \
                     python-xattr python-eventlet \
                     python-greenlet python-pastedeploy \
                     python-netifaces python-pip python-dnspython \
                     python-mock
```

- On `CentOS` (requires additional repositories):

```
sudo yum update
sudo yum install epel-release
sudo yum-config-manager --enable epel extras
sudo yum install centos-release-openstack-train
sudo yum install curl gcc memcached rsync sqlite xfsprogs git-core \
                 libffi-devel xinetd liberasurecode-devel \
                 openssl-devel python-setuptools \
                 python-coverage python-devel python-nose \
                 pyxattr python-eventlet \
                 python-greenlet python-paste-deploy \
                 python-netifaces python-pip python-dns \
                 python-mock
```

- On `Fedora`:

```
sudo dnf update
sudo dnf install curl gcc memcached rsync-daemon sqlite xfsprogs git-core␣
↪\
                 libffi-devel xinetd liberasurecode-devel \
                 openssl-devel python-setuptools \
                 python-coverage python-devel python-nose \
                 pyxattr python-eventlet \
                 python-greenlet python-paste-deploy \
                 python-netifaces python-pip python-dns \
                 python-mock
```

- On `OpenSuse`:

```
sudo zypper install curl gcc memcached rsync sqlite3 xfsprogs git-core \
                    libffi-devel liberasurecode-devel python2-setuptools \
                    libopenssl-devel
sudo zypper install python2-coverage python-devel python2-nose \
                    python-xattr python-eventlet python2-greenlet \
                    python2-netifaces python2-pip python2-dnspython \
                    python2-mock
```

Note: This installs necessary system dependencies and *most* of the python dependencies. Later in the process setuptools/distribute or pip will install and/or upgrade packages.

4.2.4 Configuring storage

Swift requires some space on XFS filesystems to store data and run tests.

Choose either *Using a partition for storage* or *Using a loopback device for storage*.

Using a partition for storage

If you are going to use a separate partition for Swift data, be sure to add another device when creating the VM, and follow these instructions:

Note: The disk does not have to be `/dev/sdb1` (for example, it could be `/dev/vdb1`) however the mount point should still be `/mnt/sdb1`.

1. Set up a single partition on the device (this will wipe the drive):

```
sudo parted /dev/sdb mklabel msdos mkpart p xfs 0% 100%
```

2. Create an XFS file system on the partition:

```
sudo mkfs.xfs /dev/sdb1
```

3. Find the UUID of the new partition:

```
sudo blkid
```

4. Edit `/etc/fstab` and add:

```
UUID="<UUID-from-output-above>" /mnt/sdb1 xfs noatime 0 0
```

5. Create the Swift data mount point and test that mounting works:

```
sudo mkdir /mnt/sdb1
sudo mount -a
```

6. Next, skip to *Common Post-Device Setup*.

Using a loopback device for storage

If you want to use a loopback device instead of another partition, follow these instructions:

1. Create the file for the loopback device:

```
sudo mkdir -p /srv
sudo truncate -s 1GB /srv/swift-disk
sudo mkfs.xfs /srv/swift-disk
```

   Modify size specified in the `truncate` command to make a larger or smaller partition as needed.

2. Edit */etc/fstab* and add:

```
/srv/swift-disk /mnt/sdb1 xfs loop,noatime 0 0
```

3. Create the Swift data mount point and test that mounting works:

```
sudo mkdir /mnt/sdb1
sudo mount -a
```

Common Post-Device Setup

1. Create the individualized data links:

```
sudo mkdir /mnt/sdb1/1 /mnt/sdb1/2 /mnt/sdb1/3 /mnt/sdb1/4
sudo chown ${USER}:${USER} /mnt/sdb1/*
for x in {1..4}; do sudo ln -s /mnt/sdb1/$x /srv/$x; done
sudo mkdir -p /srv/1/node/sdb1 /srv/1/node/sdb5 \
              /srv/2/node/sdb2 /srv/2/node/sdb6 \
              /srv/3/node/sdb3 /srv/3/node/sdb7 \
              /srv/4/node/sdb4 /srv/4/node/sdb8
sudo mkdir -p /var/run/swift
sudo mkdir -p /var/cache/swift /var/cache/swift2 \
              /var/cache/swift3 /var/cache/swift4
sudo chown -R ${USER}:${USER} /var/run/swift
sudo chown -R ${USER}:${USER} /var/cache/swift*
# **Make sure to include the trailing slash after /srv/$x/**
for x in {1..4}; do sudo chown -R ${USER}:${USER} /srv/$x/; done
```

Note: We create the mount points and mount the loopback file under /mnt/sdb1. This file will contain one directory per simulated Swift node, each owned by the current Swift user.

We then create symlinks to these directories under /srv. If the disk sdb or loopback file is unmounted, files will not be written under /srv/*, because the symbolic link destination /mnt/sdb1/* will not exist. This prevents disk sync operations from writing to the root partition in the event a drive is unmounted.

---

2. Restore appropriate permissions on reboot.

   - On traditional Linux systems, add the following lines to `/etc/rc.local` (before the `exit 0`):

```
mkdir -p /var/cache/swift /var/cache/swift2 /var/cache/swift3 /var/
↪cache/swift4
chown <your-user-name>:<your-group-name> /var/cache/swift*
mkdir -p /var/run/swift
chown <your-user-name>:<your-group-name> /var/run/swift
```

- On CentOS and Fedora we can use systemd (rc.local is deprecated):

```
cat << EOF |sudo tee /etc/tmpfiles.d/swift.conf
d /var/cache/swift 0755 ${USER} ${USER} - -
d /var/cache/swift2 0755 ${USER} ${USER} - -
d /var/cache/swift3 0755 ${USER} ${USER} - -
d /var/cache/swift4 0755 ${USER} ${USER} - -
d /var/run/swift 0755 ${USER} ${USER} - -
EOF
```

- On OpenSuse place the lines in `/etc/init.d/boot.local`.

Note: On some systems the rc file might need to be an executable shell script.

Creating an XFS tmp dir

Tests require having a directory available on an XFS filesystem. By default the tests use `/tmp`, however this can be pointed elsewhere with the `TMPDIR` environment variable.

Note: If your root filesystem is XFS, you can skip this section if `/tmp` is just a directory and not a mounted tmpfs. Or you could simply point to any existing directory owned by your user by specifying it with the `TMPDIR` environment variable.

If your root filesystem is not XFS, you should create a loopback device, format it with XFS and mount it. You can mount it over `/tmp` or to another location and specify it with the `TMPDIR` environment variable.

- Create the file for the tmp loopback device:

```
sudo mkdir -p /srv
sudo truncate -s 1GB /srv/swift-tmp  # create 1GB file for XFS in /srv
sudo mkfs.xfs /srv/swift-tmp
```

- To mount the tmp loopback device at `/tmp`, do the following:

```
sudo mount -o loop,noatime /srv/swift-tmp /tmp
sudo chmod -R 1777 /tmp
```

  - To persist this, edit and add the following to `/etc/fstab`:

```
/srv/swift-tmp /tmp xfs rw,noatime,attr2,inode64,noquota 0 0
```

- To mount the tmp loopback at an alternate location (for example, `/mnt/tmp`), do the following:

```
sudo mkdir -p /mnt/tmp
sudo mount -o loop,noatime /srv/swift-tmp /mnt/tmp
sudo chown ${USER}:${USER} /mnt/tmp
```

- To persist this, edit and add the following to `/etc/fstab`:

```
/srv/swift-tmp /mnt/tmp xfs rw,noatime,attr2,inode64,noquota 0 0
```

- Set your `TMPDIR` environment dir so that Swift looks in the right location:

```
export TMPDIR=/mnt/tmp
echo "export TMPDIR=/mnt/tmp" >> $HOME/.bashrc
```

4.2.5 Getting the code

1. Check out the python-swiftclient repo:

```
cd $HOME; git clone https://opendev.org/openstack/python-swiftclient.git
```

2. Build a development installation of python-swiftclient:

```
cd $HOME/python-swiftclient; sudo python setup.py develop; cd -
```

Ubuntu 12.04 users need to install python-swiftclients dependencies before the installation of python-swiftclient. This is due to a bug in an older version of setup tools:

```
cd $HOME/python-swiftclient; sudo pip install -r requirements.txt; sudo␣
↪python setup.py develop; cd -
```

3. Check out the Swift repo:

```
git clone https://github.com/openstack/swift.git
```

4. Build a development installation of Swift:

```
cd $HOME/swift; sudo pip install --no-binary cryptography -r requirements.
↪txt; sudo python setup.py develop; cd -
```

Note: Due to a difference in how `libssl.so` is named in OpenSuse vs. other Linux distros the wheel/binary wont work; thus we use `--no-binary cryptography` to build `cryptography` locally.

Fedora users might have to perform the following if development installation of Swift fails:

```
sudo pip install -U xattr
```

5. Install Swifts test dependencies:

```
cd $HOME/swift; sudo pip install -r test-requirements.txt
```

4.2.6 Setting up rsync

1. Create `/etc/rsyncd.conf`:

```
sudo cp $HOME/swift/doc/saio/rsyncd.conf /etc/
sudo sed -i "s/<your-user-name>/${USER}/" /etc/rsyncd.conf
```

Here is the default `rsyncd.conf` file contents maintained in the repo that is copied and fixed up above:

```
uid = <your-user-name>
gid = <your-user-name>
log file = /var/log/rsyncd.log
pid file = /var/run/rsyncd.pid
address = 0.0.0.0

[account6212]
max connections = 25
path = /srv/1/node/
read only = false
lock file = /var/lock/account6212.lock

[account6222]
max connections = 25
path = /srv/2/node/
read only = false
lock file = /var/lock/account6222.lock

[account6232]
max connections = 25
path = /srv/3/node/
read only = false
lock file = /var/lock/account6232.lock

[account6242]
max connections = 25
path = /srv/4/node/
read only = false
lock file = /var/lock/account6242.lock

[container6211]
max connections = 25
path = /srv/1/node/
read only = false
lock file = /var/lock/container6211.lock

[container6221]
max connections = 25
path = /srv/2/node/
read only = false
lock file = /var/lock/container6221.lock
```

(continues on next page)

(continued from previous page)

```
[container6231]
max connections = 25
path = /srv/3/node/
read only = false
lock file = /var/lock/container6231.lock

[container6241]
max connections = 25
path = /srv/4/node/
read only = false
lock file = /var/lock/container6241.lock

[object6210]
max connections = 25
path = /srv/1/node/
read only = false
lock file = /var/lock/object6210.lock

[object6220]
max connections = 25
path = /srv/2/node/
read only = false
lock file = /var/lock/object6220.lock

[object6230]
max connections = 25
path = /srv/3/node/
read only = false
lock file = /var/lock/object6230.lock

[object6240]
max connections = 25
path = /srv/4/node/
read only = false
lock file = /var/lock/object6240.lock
```

2. Enable rsync daemon

   - On Ubuntu, edit the following line in `/etc/default/rsync`:

     ```
     RSYNC_ENABLE=true
     ```

     Note: You might have to create the file to perform the edits.

   - On CentOS and Fedora, enable the systemd service:

     ```
     sudo systemctl enable rsyncd
     ```

   - On OpenSuse, nothing needs to happen here.

3. On platforms with SELinux in `Enforcing` mode, either set to `Permissive`:

```
sudo setenforce Permissive
sudo sed -i 's/^SELINUX=.*/SELINUX=permissive/g' /etc/selinux/config
```

   Or just allow rsync full access:

```
sudo setsebool -P rsync_full_access 1
```

4. Start the rsync daemon

   - On Ubuntu 14.04, run:

```
sudo service rsync restart
```

   - On Ubuntu 16.04, run:

```
sudo systemctl enable rsync
sudo systemctl start rsync
```

   - On CentOS, Fedora and OpenSuse, run:

```
sudo systemctl start rsyncd
```

   - On other xinetd based systems simply run:

```
sudo service xinetd restart
```

5. Verify rsync is accepting connections for all servers:

```
rsync rsync://pub@localhost/
```

   You should see the following output from the above command:

```
account6212
account6222
account6232
account6242
container6211
container6221
container6231
container6241
object6210
object6220
object6230
object6240
```

4.2.7 Starting memcached

On non-Ubuntu distros you need to ensure memcached is running:

```
sudo service memcached start
sudo chkconfig memcached on
```

or:

```
sudo systemctl enable memcached
sudo systemctl start memcached
```

The tempauth middleware stores tokens in memcached. If memcached is not running, tokens cannot be validated, and accessing Swift becomes impossible.

4.2.8 Optional: Setting up rsyslog for individual logging

Fedora and OpenSuse may not have rsyslog installed, in which case you will need to install it if you want to use individual logging.

1. Install rsyslogd
   - On Fedora:

     ```
     sudo dnf install rsyslog
     ```

   - On OpenSuse:

     ```
     sudo zypper install rsyslog
     ```

2. Install the Swift rsyslogd configuration:

   ```
   sudo cp $HOME/swift/doc/saio/rsyslog.d/10-swift.conf /etc/rsyslog.d/
   ```

   Be sure to review that conf file to determine if you want all the logs in one file vs. all the logs separated out, and if you want hourly logs for stats processing. For convenience, we provide its default contents below:

   ```
   # Uncomment the following to have a log containing all logs together
   #local1,local2,local3,local4,local5.*   /var/log/swift/all.log

   # Uncomment the following to have hourly proxy logs for stats processing
   #$template HourlyProxyLog,"/var/log/swift/hourly/%$YEAR%%$MONTH%%$DAY%%
   ↪$HOUR%"
   #local1.*;local1.!notice ?HourlyProxyLog

   local1.*;local1.!notice /var/log/swift/proxy.log
   local1.notice           /var/log/swift/proxy.error
   local1.*                ~

   local2.*;local2.!notice /var/log/swift/storage1.log
   local2.notice           /var/log/swift/storage1.error
   ```

(continues on next page)

(continued from previous page)

```
local2.*                ~

local3.*;local3.!notice /var/log/swift/storage2.log
local3.notice           /var/log/swift/storage2.error
local3.*                ~

local4.*;local4.!notice /var/log/swift/storage3.log
local4.notice           /var/log/swift/storage3.error
local4.*                ~

local5.*;local5.!notice /var/log/swift/storage4.log
local5.notice           /var/log/swift/storage4.error
local5.*                ~

local6.*;local6.!notice /var/log/swift/expirer.log
local6.notice           /var/log/swift/expirer.error
local6.*                ~
```

3. Edit `/etc/rsyslog.conf` and make the following change (usually in the GLOBAL DIRECTIVES section):

```
$PrivDropToGroup adm
```

4. If using hourly logs (see above) perform:

```
sudo mkdir -p /var/log/swift/hourly
```

   Otherwise perform:

```
sudo mkdir -p /var/log/swift
```

5. Setup the logging directory and start syslog:

   - On Ubuntu:

```
sudo chown -R syslog.adm /var/log/swift
sudo chmod -R g+w /var/log/swift
sudo service rsyslog restart
```

   - On CentOS, Fedora and OpenSuse:

```
sudo chown -R root:adm /var/log/swift
sudo chmod -R g+w /var/log/swift
sudo systemctl restart rsyslog
sudo systemctl enable rsyslog
```

4.2.9 Configuring each node

After performing the following steps, be sure to verify that Swift has access to resulting configuration files (sample configuration files are provided with all defaults in line-by-line comments).

1. Optionally remove an existing swift directory:

```
sudo rm -rf /etc/swift
```

2. Populate the `/etc/swift` directory itself:

```
cd $HOME/swift/doc; sudo cp -r saio/swift /etc/swift; cd -
sudo chown -R ${USER}:${USER} /etc/swift
```

3. Update <your-user-name> references in the Swift config files:

```
find /etc/swift/ -name \*.conf | xargs sudo sed -i "s/<your-user-name>/$
↪{USER}/"
```

The contents of the configuration files provided by executing the above commands are as follows:

1. `/etc/swift/swift.conf`

```
[swift-hash]
# random unique strings that can never change (DO NOT LOSE)
# Use only printable chars (python -c "import string; print(string.
↪printable)")
swift_hash_path_prefix = changeme
swift_hash_path_suffix = changeme

[storage-policy:0]
name = gold
policy_type = replication
default = yes

[storage-policy:1]
name = silver
policy_type = replication

[storage-policy:2]
name = ec42
policy_type = erasure_coding
ec_type = liberasurecode_rs_vand
ec_num_data_fragments = 4
ec_num_parity_fragments = 2
```

2. `/etc/swift/proxy-server.conf`

```
[DEFAULT]
bind_ip = 127.0.0.1
bind_port = 8080
workers = 1
user = <your-user-name>
```

(continues on next page)

(continued from previous page)

```
log_facility = LOG_LOCAL1
eventlet_debug = true

[pipeline:main]
# Yes, proxy-logging appears twice. This is so that
# middleware-originated requests get logged too.
pipeline = catch_errors gatekeeper healthcheck proxy-logging cache etag-
↪quoter listing_formats bulk tempurl ratelimit crossdomain container_
↪sync tempauth staticweb copy container-quotas account-quotas slo dlo␣
↪versioned_writes symlink proxy-logging proxy-server

[filter:catch_errors]
use = egg:swift#catch_errors

[filter:healthcheck]
use = egg:swift#healthcheck

[filter:proxy-logging]
use = egg:swift#proxy_logging

[filter:bulk]
use = egg:swift#bulk

[filter:ratelimit]
use = egg:swift#ratelimit

[filter:crossdomain]
use = egg:swift#crossdomain

[filter:dlo]
use = egg:swift#dlo

[filter:slo]
use = egg:swift#slo
allow_async_delete = True

[filter:container_sync]
use = egg:swift#container_sync
current = //saio/saio_endpoint

[filter:tempurl]
use = egg:swift#tempurl

[filter:tempauth]
use = egg:swift#tempauth
user_admin_admin = admin .admin .reseller_admin
user_test_tester = testing .admin
user_test_tester2 = testing2 .admin
user_test_tester3 = testing3
```

(continues on next page)

(continued from previous page)

```
user_test2_tester2 = testing2 .admin

[filter:staticweb]
use = egg:swift#staticweb

[filter:account-quotas]
use = egg:swift#account_quotas

[filter:container-quotas]
use = egg:swift#container_quotas

[filter:cache]
use = egg:swift#memcache

[filter:etag-quoter]
use = egg:swift#etag_quoter
enable_by_default = false

[filter:gatekeeper]
use = egg:swift#gatekeeper

[filter:versioned_writes]
use = egg:swift#versioned_writes
allow_versioned_writes = true
allow_object_versioning = true

[filter:copy]
use = egg:swift#copy

[filter:listing_formats]
use = egg:swift#listing_formats

[filter:domain_remap]
use = egg:swift#domain_remap

[filter:symlink]
use = egg:swift#symlink

# To enable, add the s3api middleware to the pipeline before tempauth
[filter:s3api]
use = egg:swift#s3api
s3_acl = yes
check_bucket_owner = yes
cors_preflight_allow_origin = *

# Example to create root secret: `openssl rand -base64 32`
[filter:keymaster]
use = egg:swift#keymaster
encryption_root_secret = changeme/changeme/changeme/changeme/change/=
```

(continues on next page)

(continued from previous page)

```
# To enable use of encryption add both middlewares to pipeline, example:
# <other middleware> keymaster encryption proxy-logging proxy-server
[filter:encryption]
use = egg:swift#encryption

[app:proxy-server]
use = egg:swift#proxy
allow_account_management = true
account_autocreate = true
```

3. /etc/swift/object-expirer.conf

```
[DEFAULT]
# swift_dir = /etc/swift
user = <your-user-name>
# You can specify default log routing here if you want:
log_name = object-expirer
log_facility = LOG_LOCAL6
log_level = INFO
#log_address = /dev/log
#
# comma separated list of functions to call to setup custom log handlers.
# functions get passed: conf, name, log_to_console, log_route, fmt,␣
↪logger,
# adapted_logger
# log_custom_handlers =
#
# If set, log_udp_host will override log_address
# log_udp_host =
# log_udp_port = 514
#
# You can enable StatsD logging here:
# log_statsd_host =
# log_statsd_port = 8125
# log_statsd_default_sample_rate = 1.0
# log_statsd_sample_rate_factor = 1.0
# log_statsd_metric_prefix =

[object-expirer]
interval = 300
# report_interval = 300
# concurrency is the level of concurrency to use to do the work, this␣
↪value
# must be set to at least 1
# concurrency = 1
# processes is how many parts to divide the work into, one part per␣
↪process
#   that will be doing the work
```

(continues on next page)

(continued from previous page)

```
# processes set 0 means that a single process will be doing all the work
# processes can also be specified on the command line and will override␣
↪the
#   config value
# processes = 0
# process is which of the parts a particular process will work on
# process can also be specified on the command line and will override the␣
↪config
#   value
# process is "zero based", if you want to use 3 processes, you should run
#  processes with process set to 0, 1, and 2
# process = 0

[pipeline:main]
pipeline = catch_errors cache proxy-server

[app:proxy-server]
use = egg:swift#proxy
# See proxy-server.conf-sample for options

[filter:cache]
use = egg:swift#memcache
# See proxy-server.conf-sample for options

[filter:catch_errors]
use = egg:swift#catch_errors
# See proxy-server.conf-sample for options
```

4. /etc/swift/container-sync-realms.conf

```
[saio]
key = changeme
key2 = changeme
cluster_saio_endpoint = http://127.0.0.1:8080/v1/
```

5. /etc/swift/account-server/1.conf

```
[DEFAULT]
devices = /srv/1/node
mount_check = false
disable_fallocate = true
bind_ip = 127.0.0.1
bind_port = 6212
workers = 1
user = <your-user-name>
log_facility = LOG_LOCAL2
recon_cache_path = /var/cache/swift
eventlet_debug = true

[pipeline:main]
```

(continues on next page)

(continued from previous page)

```
pipeline = healthcheck recon account-server

[app:account-server]
use = egg:swift#account

[filter:recon]
use = egg:swift#recon

[filter:healthcheck]
use = egg:swift#healthcheck

[account-replicator]
rsync_module = {replication_ip}::account{replication_port}

[account-auditor]

[account-reaper]
```

6. /etc/swift/container-server/1.conf

```
[DEFAULT]
devices = /srv/1/node
mount_check = false
disable_fallocate = true
bind_ip = 127.0.0.1
bind_port = 6211
workers = 1
user = <your-user-name>
log_facility = LOG_LOCAL2
recon_cache_path = /var/cache/swift
eventlet_debug = true

[pipeline:main]
pipeline = healthcheck recon container-server

[app:container-server]
use = egg:swift#container

[filter:recon]
use = egg:swift#recon

[filter:healthcheck]
use = egg:swift#healthcheck

[container-replicator]
rsync_module = {replication_ip}::container{replication_port}

[container-updater]
```

(continues on next page)

(continued from previous page)

```
[container-auditor]

[container-sync]

[container-sharder]
auto_shard = true
rsync_module = {replication_ip}::container{replication_port}
# This is intentionally much smaller than the default of 1,000,000 so␣
↪tests
# can run in a reasonable amount of time
shard_container_threshold = 100
# The probe tests make explicit assumptions about the batch sizes
shard_scanner_batch_size = 10
cleave_batch_size = 2
```

7. /etc/swift/container-reconciler/1.conf

```
[DEFAULT]
# swift_dir = /etc/swift
user = <your-user-name>
# You can specify default log routing here if you want:
# log_name = swift
# log_facility = LOG_LOCAL0
# log_level = INFO
# log_address = /dev/log
#
# comma separated list of functions to call to setup custom log handlers.
# functions get passed: conf, name, log_to_console, log_route, fmt,␣
↪logger,
# adapted_logger
# log_custom_handlers =
#
# If set, log_udp_host will override log_address
# log_udp_host =
# log_udp_port = 514
#
# You can enable StatsD logging here:
# log_statsd_host =
# log_statsd_port = 8125
# log_statsd_default_sample_rate = 1.0
# log_statsd_sample_rate_factor = 1.0
# log_statsd_metric_prefix =

[container-reconciler]
# reclaim_age = 604800
# interval = 300
# request_tries = 3
processes = 4
process = 0
```

(continues on next page)

(continued from previous page)

```
[pipeline:main]
pipeline = catch_errors proxy-logging cache proxy-server

[app:proxy-server]
use = egg:swift#proxy
# See proxy-server.conf-sample for options

[filter:cache]
use = egg:swift#memcache
# See proxy-server.conf-sample for options

[filter:proxy-logging]
use = egg:swift#proxy_logging

[filter:catch_errors]
use = egg:swift#catch_errors
# See proxy-server.conf-sample for options
```

8. /etc/swift/object-server/1.conf

```
[DEFAULT]
devices = /srv/1/node
mount_check = false
disable_fallocate = true
bind_ip = 127.0.0.1
bind_port = 6210
workers = 1
user = <your-user-name>
log_facility = LOG_LOCAL2
recon_cache_path = /var/cache/swift
eventlet_debug = true

[pipeline:main]
pipeline = healthcheck recon object-server

[app:object-server]
use = egg:swift#object

[filter:recon]
use = egg:swift#recon

[filter:healthcheck]
use = egg:swift#healthcheck

[object-replicator]
rsync_module = {replication_ip}::object{replication_port}

[object-reconstructor]
```

(continues on next page)

(continued from previous page)

```
[object-updater]

[object-auditor]

[object-relinker]
```

9. /etc/swift/account-server/2.conf

```
[DEFAULT]
devices = /srv/2/node
mount_check = false
disable_fallocate = true
bind_ip = 127.0.0.2
bind_port = 6222
workers = 1
user = <your-user-name>
log_facility = LOG_LOCAL3
recon_cache_path = /var/cache/swift2
eventlet_debug = true

[pipeline:main]
pipeline = healthcheck recon account-server

[app:account-server]
use = egg:swift#account

[filter:recon]
use = egg:swift#recon

[filter:healthcheck]
use = egg:swift#healthcheck

[account-replicator]
rsync_module = {replication_ip}::account{replication_port}

[account-auditor]

[account-reaper]
```

10. /etc/swift/container-server/2.conf

```
[DEFAULT]
devices = /srv/2/node
mount_check = false
disable_fallocate = true
bind_ip = 127.0.0.2
bind_port = 6221
workers = 1
user = <your-user-name>
```

(continues on next page)

(continued from previous page)

```
log_facility = LOG_LOCAL3
recon_cache_path = /var/cache/swift2
eventlet_debug = true

[pipeline:main]
pipeline = healthcheck recon container-server

[app:container-server]
use = egg:swift#container

[filter:recon]
use = egg:swift#recon

[filter:healthcheck]
use = egg:swift#healthcheck

[container-replicator]
rsync_module = {replication_ip}::container{replication_port}

[container-updater]

[container-auditor]

[container-sync]

[container-sharder]
auto_shard = true
rsync_module = {replication_ip}::container{replication_port}
# This is intentionally much smaller than the default of 1,000,000 so␣
↪tests
# can run in a reasonable amount of time
shard_container_threshold = 100
# The probe tests make explicit assumptions about the batch sizes
shard_scanner_batch_size = 10
cleave_batch_size = 2
```

11. /etc/swift/container-reconciler/2.conf

```
[DEFAULT]
# swift_dir = /etc/swift
user = <your-user-name>
# You can specify default log routing here if you want:
# log_name = swift
# log_facility = LOG_LOCAL0
# log_level = INFO
# log_address = /dev/log
#
# comma separated list of functions to call to setup custom log handlers.
# functions get passed: conf, name, log_to_console, log_route, fmt,␣
↪logger,
```

(continues on next page)

(continued from previous page)

```
# adapted_logger
# log_custom_handlers =
#
# If set, log_udp_host will override log_address
# log_udp_host =
# log_udp_port = 514
#
# You can enable StatsD logging here:
# log_statsd_host =
# log_statsd_port = 8125
# log_statsd_default_sample_rate = 1.0
# log_statsd_sample_rate_factor = 1.0
# log_statsd_metric_prefix =

[container-reconciler]
# reclaim_age = 604800
# interval = 300
# request_tries = 3
processes = 4
process = 1

[pipeline:main]
pipeline = catch_errors proxy-logging cache proxy-server

[app:proxy-server]
use = egg:swift#proxy
# See proxy-server.conf-sample for options

[filter:cache]
use = egg:swift#memcache
# See proxy-server.conf-sample for options

[filter:proxy-logging]
use = egg:swift#proxy_logging

[filter:catch_errors]
use = egg:swift#catch_errors
# See proxy-server.conf-sample for options
```

12. /etc/swift/object-server/2.conf

```
[DEFAULT]
devices = /srv/2/node
mount_check = false
disable_fallocate = true
bind_ip = 127.0.0.2
bind_port = 6220
workers = 1
user = <your-user-name>
```

(continues on next page)

(continued from previous page)

```
log_facility = LOG_LOCAL3
recon_cache_path = /var/cache/swift2
eventlet_debug = true

[pipeline:main]
pipeline = healthcheck recon object-server

[app:object-server]
use = egg:swift#object

[filter:recon]
use = egg:swift#recon

[filter:healthcheck]
use = egg:swift#healthcheck

[object-replicator]
rsync_module = {replication_ip}::object{replication_port}

[object-reconstructor]

[object-updater]

[object-auditor]

[object-relinker]
```

13. /etc/swift/account-server/3.conf

```
[DEFAULT]
devices = /srv/3/node
mount_check = false
disable_fallocate = true
bind_ip = 127.0.0.3
bind_port = 6232
workers = 1
user = <your-user-name>
log_facility = LOG_LOCAL4
recon_cache_path = /var/cache/swift3
eventlet_debug = true

[pipeline:main]
pipeline = healthcheck recon account-server

[app:account-server]
use = egg:swift#account

[filter:recon]
use = egg:swift#recon
```

(continues on next page)

(continued from previous page)

```
[filter:healthcheck]
use = egg:swift#healthcheck

[account-replicator]
rsync_module = {replication_ip}::account{replication_port}

[account-auditor]

[account-reaper]
```

14. /etc/swift/container-server/3.conf

```
[DEFAULT]
devices = /srv/3/node
mount_check = false
disable_fallocate = true
bind_ip = 127.0.0.3
bind_port = 6231
workers = 1
user = <your-user-name>
log_facility = LOG_LOCAL4
recon_cache_path = /var/cache/swift3
eventlet_debug = true

[pipeline:main]
pipeline = healthcheck recon container-server

[app:container-server]
use = egg:swift#container

[filter:recon]
use = egg:swift#recon

[filter:healthcheck]
use = egg:swift#healthcheck

[container-replicator]
rsync_module = {replication_ip}::container{replication_port}

[container-updater]

[container-auditor]

[container-sync]

[container-sharder]
auto_shard = true
rsync_module = {replication_ip}::container{replication_port}
```

(continues on next page)

(continued from previous page)

```
# This is intentionally much smaller than the default of 1,000,000 so␣
↪tests
# can run in a reasonable amount of time
shard_container_threshold = 100
# The probe tests make explicit assumptions about the batch sizes
shard_scanner_batch_size = 10
cleave_batch_size = 2
```

15. /etc/swift/container-reconciler/3.conf

```
[DEFAULT]
# swift_dir = /etc/swift
user = <your-user-name>
# You can specify default log routing here if you want:
# log_name = swift
# log_facility = LOG_LOCAL0
# log_level = INFO
# log_address = /dev/log
#
# comma separated list of functions to call to setup custom log handlers.
# functions get passed: conf, name, log_to_console, log_route, fmt,␣
↪logger,
# adapted_logger
# log_custom_handlers =
#
# If set, log_udp_host will override log_address
# log_udp_host =
# log_udp_port = 514
#
# You can enable StatsD logging here:
# log_statsd_host =
# log_statsd_port = 8125
# log_statsd_default_sample_rate = 1.0
# log_statsd_sample_rate_factor = 1.0
# log_statsd_metric_prefix =

[container-reconciler]
# reclaim_age = 604800
# interval = 300
# request_tries = 3
processes = 4
process = 2

[pipeline:main]
pipeline = catch_errors proxy-logging cache proxy-server

[app:proxy-server]
use = egg:swift#proxy
# See proxy-server.conf-sample for options
```

(continues on next page)

(continued from previous page)

```
[filter:cache]
use = egg:swift#memcache
# See proxy-server.conf-sample for options

[filter:proxy-logging]
use = egg:swift#proxy_logging

[filter:catch_errors]
use = egg:swift#catch_errors
# See proxy-server.conf-sample for options
```

16. /etc/swift/object-server/3.conf

```
[DEFAULT]
devices = /srv/3/node
mount_check = false
disable_fallocate = true
bind_ip = 127.0.0.3
bind_port = 6230
workers = 1
user = <your-user-name>
log_facility = LOG_LOCAL4
recon_cache_path = /var/cache/swift3
eventlet_debug = true

[pipeline:main]
pipeline = healthcheck recon object-server

[app:object-server]
use = egg:swift#object

[filter:recon]
use = egg:swift#recon

[filter:healthcheck]
use = egg:swift#healthcheck

[object-replicator]
rsync_module = {replication_ip}::object{replication_port}

[object-reconstructor]

[object-updater]

[object-auditor]

[object-relinker]
```

17. /etc/swift/account-server/4.conf

```
[DEFAULT]
devices = /srv/4/node
mount_check = false
disable_fallocate = true
bind_ip = 127.0.0.4
bind_port = 6242
workers = 1
user = <your-user-name>
log_facility = LOG_LOCAL5
recon_cache_path = /var/cache/swift4
eventlet_debug = true

[pipeline:main]
pipeline = healthcheck recon account-server

[app:account-server]
use = egg:swift#account

[filter:recon]
use = egg:swift#recon

[filter:healthcheck]
use = egg:swift#healthcheck

[account-replicator]
rsync_module = {replication_ip}::account{replication_port}

[account-auditor]

[account-reaper]
```

18. /etc/swift/container-server/4.conf

```
[DEFAULT]
devices = /srv/4/node
mount_check = false
disable_fallocate = true
bind_ip = 127.0.0.4
bind_port = 6241
workers = 1
user = <your-user-name>
log_facility = LOG_LOCAL5
recon_cache_path = /var/cache/swift4
eventlet_debug = true

[pipeline:main]
pipeline = healthcheck recon container-server

[app:container-server]
use = egg:swift#container
```

(continues on next page)

(continued from previous page)

```
[filter:recon]
use = egg:swift#recon

[filter:healthcheck]
use = egg:swift#healthcheck

[container-replicator]
rsync_module = {replication_ip}::container{replication_port}

[container-updater]

[container-auditor]

[container-sync]

[container-sharder]
auto_shard = true
rsync_module = {replication_ip}::container{replication_port}
# This is intentionally much smaller than the default of 1,000,000 so␣
↪tests
# can run in a reasonable amount of time
shard_container_threshold = 100
# The probe tests make explicit assumptions about the batch sizes
shard_scanner_batch_size = 10
cleave_batch_size = 2
```

19. /etc/swift/container-reconciler/4.conf

```
[DEFAULT]
# swift_dir = /etc/swift
user = <your-user-name>
# You can specify default log routing here if you want:
# log_name = swift
# log_facility = LOG_LOCAL0
# log_level = INFO
# log_address = /dev/log
#
# comma separated list of functions to call to setup custom log handlers.
# functions get passed: conf, name, log_to_console, log_route, fmt,␣
↪logger,
# adapted_logger
# log_custom_handlers =
#
# If set, log_udp_host will override log_address
# log_udp_host =
# log_udp_port = 514
#
# You can enable StatsD logging here:
```

(continues on next page)

(continued from previous page)

```
# log_statsd_host =
# log_statsd_port = 8125
# log_statsd_default_sample_rate = 1.0
# log_statsd_sample_rate_factor = 1.0
# log_statsd_metric_prefix =

[container-reconciler]
# reclaim_age = 604800
# interval = 300
# request_tries = 3
processes = 4
process = 3

[pipeline:main]
pipeline = catch_errors proxy-logging cache proxy-server

[app:proxy-server]
use = egg:swift#proxy
# See proxy-server.conf-sample for options

[filter:cache]
use = egg:swift#memcache
# See proxy-server.conf-sample for options

[filter:proxy-logging]
use = egg:swift#proxy_logging

[filter:catch_errors]
use = egg:swift#catch_errors
# See proxy-server.conf-sample for options
```

20. /etc/swift/object-server/4.conf

```
[DEFAULT]
devices = /srv/4/node
mount_check = false
disable_fallocate = true
bind_ip = 127.0.0.4
bind_port = 6240
workers = 1
user = <your-user-name>
log_facility = LOG_LOCAL5
recon_cache_path = /var/cache/swift4
eventlet_debug = true

[pipeline:main]
pipeline = healthcheck recon object-server

[app:object-server]
```

(continues on next page)

(continued from previous page)

```
use = egg:swift#object

[filter:recon]
use = egg:swift#recon

[filter:healthcheck]
use = egg:swift#healthcheck

[object-replicator]
rsync_module = {replication_ip}::object{replication_port}

[object-reconstructor]

[object-updater]

[object-auditor]

[object-relinker]
```

4.2.10 Setting up scripts for running Swift

1. Copy the SAIO scripts for resetting the environment:

```
mkdir -p $HOME/bin
cd $HOME/swift/doc; cp saio/bin/* $HOME/bin; cd -
chmod +x $HOME/bin/*
```

2. Edit the `$HOME/bin/resetswift` script

   The template `resetswift` script looks like the following:

```
#!/bin/bash

set -e

swift-init all kill
swift-orphans -a 0 -k KILL

# Remove the following line if you did not set up rsyslog for individual␣
↪logging:
sudo find /var/log/swift -type f -exec rm -f {} \;
if cut -d' ' -f2 /proc/mounts | grep -q /mnt/sdb1 ; then
    sudo umount /mnt/sdb1
fi
# If you are using a loopback device set SAIO_BLOCK_DEVICE to "/srv/swift-
↪disk"
sudo mkfs.xfs -f ${SAIO_BLOCK_DEVICE:-/dev/sdb1}
sudo mount /mnt/sdb1
sudo mkdir /mnt/sdb1/1 /mnt/sdb1/2 /mnt/sdb1/3 /mnt/sdb1/4
```

(continues on next page)

(continued from previous page)

```
sudo chown ${USER}:${USER} /mnt/sdb1/*
mkdir -p /srv/1/node/sdb1 /srv/1/node/sdb5 \
         /srv/2/node/sdb2 /srv/2/node/sdb6 \
         /srv/3/node/sdb3 /srv/3/node/sdb7 \
         /srv/4/node/sdb4 /srv/4/node/sdb8
sudo rm -f /var/log/debug /var/log/messages /var/log/rsyncd.log /var/log/
↪syslog
find /var/cache/swift* -type f -name *.recon -exec rm -f {} \;
if [ "`type -t systemctl`" == "file" ]; then
    sudo systemctl restart rsyslog
    sudo systemctl restart memcached
else
    sudo service rsyslog restart
    sudo service memcached restart
fi
```

If you did not set up rsyslog for individual logging, remove the `find /var/log/swift...` line:

```
sed -i "/find \/var\/log\/swift/d" $HOME/bin/resetswift
```

3. Install the sample configuration file for running tests:

```
cp $HOME/swift/test/sample.conf /etc/swift/test.conf
```

The template `test.conf` looks like the following:

```
[s3api_test]
# You just enable advanced compatibility features to pass all tests.  Add␣
↪the
# following non-default options to the s3api section of your proxy-server.
↪conf
# s3_acl = True
# check_bucket_owner = True
endpoint = http://127.0.0.1:8080
#ca_cert=/path/to/ca.crt
region = us-east-1
# First and second users should be account owners
access_key1 = test:tester
secret_key1 = testing
access_key2 = test:tester2
secret_key2 = testing2
# Third user should be unprivileged
access_key3 = test:tester3
secret_key3 = testing3

[func_test]
# Sample config for Swift with tempauth
auth_uri = http://127.0.0.1:8080/auth/v1.0
# Sample config for Swift with Keystone v2 API.
# For keystone v2 change auth_version to 2 and auth_prefix to /v2.0/.
```

(continues on next page)

(continued from previous page)

```
# And "allow_account_management" should not be set "true".
#auth_version = 3
#auth_uri = http://localhost:5000/v3/

# Used by s3api functional tests, which don't contact auth directly
#s3_storage_url = http://127.0.0.1:8080/
#s3_region = us-east-1

# Primary functional test account (needs admin access to the account)
account = test
username = tester
password = testing
s3_access_key = test:tester
s3_secret_key = testing

# User on a second account (needs admin access to the account)
account2 = test2
username2 = tester2
password2 = testing2

# User on same account as first, but without admin access
username3 = tester3
password3 = testing3
# s3api requires the same account with the primary one and different␣
↪users
# one swift owner:
s3_access_key2 = test:tester2
s3_secret_key2 = testing2
# one unprivileged:
s3_access_key3 = test:tester3
s3_secret_key3 = testing3

# Fourth user is required for keystone v3 specific tests.
# Account must be in a non-default domain.
#account4 = test4
#username4 = tester4
#password4 = testing4
#domain4 = test-domain

# Fifth user is required for service token-specific tests.
# The account must be different from the primary test account.
# The user must not have a group (tempauth) or role (keystoneauth) on
# the primary test account. The user must have a group/role that is␣
↪unique
# and not given to the primary tester and is specified in the options
# <prefix>_require_group (tempauth) or <prefix>_service_roles␣
↪(keystoneauth).
#account5 = test5
#username5 = tester5
```

(continues on next page)

(continued from previous page)

```
#password5 = testing5

# The service_prefix option is used for service token-specific tests.
# If service_prefix or username5 above is not supplied, the tests are␣
↪skipped.
# To set the value and enable the service token tests, look at the
# reseller_prefix option in /etc/swift/proxy-server.conf. There must be␣
↪at
# least two prefixes. If not, add a prefix as follows (where we add␣
↪SERVICE):
#     reseller_prefix = AUTH, SERVICE
# The service_prefix must match the <prefix> used in <prefix>_require_
↪group
# (tempauth) or <prefix>_service_roles (keystoneauth); for example:
#    SERVICE_require_group = service
#    SERVICE_service_roles = service
# Note: Do not enable service token tests if the first prefix in
# reseller_prefix is the empty prefix AND the primary functional test
# account contains an underscore.
#service_prefix = SERVICE

# Sixth user is required for access control tests.
# Account must have a role for reseller_admin_role(keystoneauth).
#account6 = test
#username6 = tester6
#password6 = testing6

collate = C

# Only necessary if a pre-existing server uses self-signed certificate
insecure = no

# Tests that are dependent on domain_remap middleware being installed␣
↪also
# require one of the domain_remap storage_domain values to be specified␣
↪here,
# otherwise those tests will be skipped.
storage_domain =

[unit_test]
fake_syslog = False

[probe_test]
# check_server_timeout = 30
# validate_rsync = false
# proxy_base_url = http://localhost:8080

[swift-constraints]
# The functional test runner will try to use the constraint values␣
↪provided in
```

(continues on next page)

(continued from previous page)

```
# the swift-constraints section of test.conf.
#
# If a constraint value does not exist in that section, or because the
# swift-constraints section does not exist, the constraints values found␣
↪in
# the /info API call (if successful) will be used.
#
# If a constraint value cannot be found in the /info results, either␣
↪because
# the /info API call failed, or a value is not present, the constraint␣
↪value
# used will fall back to those loaded by the constraints module at time␣
↪of
# import (which will attempt to load /etc/swift/swift.conf, see the
# swift.common.constraints module for more information).
#
# Note that the cluster must have "sane" values for the test suite to␣
↪pass
# (for some definition of sane).
#
#max_file_size = 5368709122
#max_meta_name_length = 128
#max_meta_value_length = 256
#max_meta_count = 90
#max_meta_overall_size = 4096
#max_header_size = 8192
#extra_header_count = 0
#max_object_name_length = 1024
#container_listing_limit = 10000
#account_listing_limit = 10000
#max_account_name_length = 256
#max_container_name_length = 256

# Newer swift versions default to strict cors mode, but older ones were␣
↪the
# opposite.
#strict_cors_mode = true
```

4.2.11 Configure environment variables for Swift

1. Add an environment variable for running tests below:

```
echo "export SWIFT_TEST_CONFIG_FILE=/etc/swift/test.conf" >> $HOME/.bashrc
```

2. Be sure that your `PATH` includes the `bin` directory:

```
echo "export PATH=${PATH}:$HOME/bin" >> $HOME/.bashrc
```

3. If you are using a loopback device for Swift Storage, add an environment var to substitute `/dev/`

sdb1 with /srv/swift-disk:

```
echo "export SAIO_BLOCK_DEVICE=/srv/swift-disk" >> $HOME/.bashrc
```

4. If you are using a device other than /dev/sdb1 for Swift storage (for example, /dev/vdb1), add an environment var to substitute it:

```
echo "export SAIO_BLOCK_DEVICE=/dev/vdb1" >> $HOME/.bashrc
```

5. If you are using a location other than /tmp for Swift tmp data (for example, /mnt/tmp), add TMPDIR environment var to set it:

```
export TMPDIR=/mnt/tmp
echo "export TMPDIR=/mnt/tmp" >> $HOME/.bashrc
```

6. Source the above environment variables into your current environment:

```
. $HOME/.bashrc
```

4.2.12 Constructing initial rings

1. Construct the initial rings using the provided script:

```
remakerings
```

The remakerings script looks like the following:

```
#!/bin/bash

set -e

cd /etc/swift

rm -f *.builder *.ring.gz backups/*.builder backups/*.ring.gz

swift-ring-builder object.builder create 10 3 1
swift-ring-builder object.builder add r1z1-127.0.0.1:6210/sdb1 1
swift-ring-builder object.builder add r1z2-127.0.0.2:6220/sdb2 1
swift-ring-builder object.builder add r1z3-127.0.0.3:6230/sdb3 1
swift-ring-builder object.builder add r1z4-127.0.0.4:6240/sdb4 1
swift-ring-builder object.builder rebalance
swift-ring-builder object-1.builder create 10 2 1
swift-ring-builder object-1.builder add r1z1-127.0.0.1:6210/sdb1 1
swift-ring-builder object-1.builder add r1z2-127.0.0.2:6220/sdb2 1
swift-ring-builder object-1.builder add r1z3-127.0.0.3:6230/sdb3 1
swift-ring-builder object-1.builder add r1z4-127.0.0.4:6240/sdb4 1
swift-ring-builder object-1.builder rebalance
swift-ring-builder object-2.builder create 10 6 1
swift-ring-builder object-2.builder add r1z1-127.0.0.1:6210/sdb1 1
swift-ring-builder object-2.builder add r1z1-127.0.0.1:6210/sdb5 1
swift-ring-builder object-2.builder add r1z2-127.0.0.2:6220/sdb2 1
```

(continues on next page)

(continued from previous page)

```
swift-ring-builder object-2.builder add r1z2-127.0.0.2:6220/sdb6 1
swift-ring-builder object-2.builder add r1z3-127.0.0.3:6230/sdb3 1
swift-ring-builder object-2.builder add r1z3-127.0.0.3:6230/sdb7 1
swift-ring-builder object-2.builder add r1z4-127.0.0.4:6240/sdb4 1
swift-ring-builder object-2.builder add r1z4-127.0.0.4:6240/sdb8 1
swift-ring-builder object-2.builder rebalance
swift-ring-builder container.builder create 10 3 1
swift-ring-builder container.builder add r1z1-127.0.0.1:6211/sdb1 1
swift-ring-builder container.builder add r1z2-127.0.0.2:6221/sdb2 1
swift-ring-builder container.builder add r1z3-127.0.0.3:6231/sdb3 1
swift-ring-builder container.builder add r1z4-127.0.0.4:6241/sdb4 1
swift-ring-builder container.builder rebalance
swift-ring-builder account.builder create 10 3 1
swift-ring-builder account.builder add r1z1-127.0.0.1:6212/sdb1 1
swift-ring-builder account.builder add r1z2-127.0.0.2:6222/sdb2 1
swift-ring-builder account.builder add r1z3-127.0.0.3:6232/sdb3 1
swift-ring-builder account.builder add r1z4-127.0.0.4:6242/sdb4 1
swift-ring-builder account.builder rebalance
```

You can expect the output from this command to produce the following. Note that 3 object rings are created in order to test storage policies and EC in the SAIO environment. The EC ring is the only one with all 8 devices. There are also two replication rings, one for 3x replication and another for 2x replication, but those rings only use 4 devices:

```
Device d0r1z1-127.0.0.1:6210R127.0.0.1:6210/sdb1_"" with 1.0 weight got␣
↪id 0
Device d1r1z2-127.0.0.2:6220R127.0.0.2:6220/sdb2_"" with 1.0 weight got␣
↪id 1
Device d2r1z3-127.0.0.3:6230R127.0.0.3:6230/sdb3_"" with 1.0 weight got␣
↪id 2
Device d3r1z4-127.0.0.4:6240R127.0.0.4:6240/sdb4_"" with 1.0 weight got␣
↪id 3
Reassigned 3072 (300.00%) partitions. Balance is now 0.00.  Dispersion is␣
↪now 0.00
Device d0r1z1-127.0.0.1:6210R127.0.0.1:6210/sdb1_"" with 1.0 weight got␣
↪id 0
Device d1r1z2-127.0.0.2:6220R127.0.0.2:6220/sdb2_"" with 1.0 weight got␣
↪id 1
Device d2r1z3-127.0.0.3:6230R127.0.0.3:6230/sdb3_"" with 1.0 weight got␣
↪id 2
Device d3r1z4-127.0.0.4:6240R127.0.0.4:6240/sdb4_"" with 1.0 weight got␣
↪id 3
Reassigned 2048 (200.00%) partitions. Balance is now 0.00.  Dispersion is␣
↪now 0.00
Device d0r1z1-127.0.0.1:6210R127.0.0.1:6210/sdb1_"" with 1.0 weight got␣
↪id 0
Device d1r1z1-127.0.0.1:6210R127.0.0.1:6210/sdb5_"" with 1.0 weight got␣
↪id 1
Device d2r1z2-127.0.0.2:6220R127.0.0.2:6220/sdb2_"" with 1.0 weight got␣
↪id 2
```

(continues on next page)

(continued from previous page)

```
Device d3r1z2-127.0.0.2:6220R127.0.0.2:6220/sdb6_"" with 1.0 weight got␣
↪id 3
Device d4r1z3-127.0.0.3:6230R127.0.0.3:6230/sdb3_"" with 1.0 weight got␣
↪id 4
Device d5r1z3-127.0.0.3:6230R127.0.0.3:6230/sdb7_"" with 1.0 weight got␣
↪id 5
Device d6r1z4-127.0.0.4:6240R127.0.0.4:6240/sdb4_"" with 1.0 weight got␣
↪id 6
Device d7r1z4-127.0.0.4:6240R127.0.0.4:6240/sdb8_"" with 1.0 weight got␣
↪id 7
Reassigned 6144 (600.00%) partitions. Balance is now 0.00.  Dispersion is␣
↪now 0.00
Device d0r1z1-127.0.0.1:6211R127.0.0.1:6211/sdb1_"" with 1.0 weight got␣
↪id 0
Device d1r1z2-127.0.0.2:6221R127.0.0.2:6221/sdb2_"" with 1.0 weight got␣
↪id 1
Device d2r1z3-127.0.0.3:6231R127.0.0.3:6231/sdb3_"" with 1.0 weight got␣
↪id 2
Device d3r1z4-127.0.0.4:6241R127.0.0.4:6241/sdb4_"" with 1.0 weight got␣
↪id 3
Reassigned 3072 (300.00%) partitions. Balance is now 0.00.  Dispersion is␣
↪now 0.00
Device d0r1z1-127.0.0.1:6212R127.0.0.1:6212/sdb1_"" with 1.0 weight got␣
↪id 0
Device d1r1z2-127.0.0.2:6222R127.0.0.2:6222/sdb2_"" with 1.0 weight got␣
↪id 1
Device d2r1z3-127.0.0.3:6232R127.0.0.3:6232/sdb3_"" with 1.0 weight got␣
↪id 2
Device d3r1z4-127.0.0.4:6242R127.0.0.4:6242/sdb4_"" with 1.0 weight got␣
↪id 3
Reassigned 3072 (300.00%) partitions. Balance is now 0.00.  Dispersion is␣
↪now 0.00
```

2. Read more about Storage Policies and your SAIO *Adding Storage Policies to an Existing SAIO*

4.2.13 Testing Swift

1. Verify the unit tests run:

```
$HOME/swift/.unittests
```

   Note that the unit tests do not require any Swift daemons running.

2. Start the main Swift daemon processes (proxy, account, container, and object):

```
startmain
```

   (The `Unable to increase file descriptor limit. Running as non-root?` warnings are expected and ok.)

   The `startmain` script looks like the following:

```
#!/bin/bash

set -e

swift-init main start
```

3. Get an `X-Storage-Url` and `X-Auth-Token`:

```
curl -v -H 'X-Storage-User: test:tester' -H 'X-Storage-Pass: testing'␣
↪http://127.0.0.1:8080/auth/v1.0
```

4. Check that you can `GET` account:

```
curl -v -H 'X-Auth-Token: <token-from-x-auth-token-above>' <url-from-x-
↪storage-url-above>
```

5. Check that `swift` command provided by the python-swiftclient package works:

```
swift -A http://127.0.0.1:8080/auth/v1.0 -U test:tester -K testing stat
```

6. Verify the functional tests run:

```
$HOME/swift/.functests
```

(Note: functional tests will first delete everything in the configured accounts.)

7. Verify the probe tests run:

```
$HOME/swift/.probetests
```

(Note: probe tests will reset your environment as they call `resetswift` for each test.)

4.2.14 Debugging Issues

If all doesnt go as planned, and tests fail, or you cant auth, or something doesnt work, here are some good starting places to look for issues:

1. Everything is logged using system facilities usually in `/var/log/syslog`, but possibly in `/var/log/messages` on e.g. Fedora so that is a good first place to look for errors (most likely python tracebacks).
2. Make sure all of the server processes are running. For the base functionality, the Proxy, Account, Container, and Object servers should be running.
3. If one of the servers are not running, and no errors are logged to syslog, it may be useful to try to start the server manually, for example: `swift-object-server /etc/swift/object-server/1.conf` will start the object server. If there are problems not showing up in syslog, then you will likely see the traceback on startup.
4. If you need to, you can turn off syslog for unit tests. This can be useful for environments where `/dev/log` is unavailable, or which cannot rate limit (unit tests generate a lot of logs very quickly). Open the file `SWIFT_TEST_CONFIG_FILE` points to, and change the value of `fake_syslog` to `True`.

5. If you encounter a `401 Unauthorized` when following Step 12 where you check that you can `GET` account, use `sudo service memcached status` and check if memcache is running. If memcache is not running, start it using `sudo service memcached start`. Once memcache is running, rerun `GET` account.

4.2.15 Known Issues

Listed here are some gotchas that you may run into when using or testing your SAIO:

1. fallocate_reserve - in most cases a SAIO doesnt have a very large XFS partition so having fallocate enabled and fallocate_reserve set can cause issues, specifically when trying to run the functional tests. For this reason fallocate has been turned off on the object-servers in the SAIO. If you want to play with the fallocate_reserve settings then know that functional tests will fail unless you change the max_file_size constraint to something more reasonable then the default (5G). Ideally youd make it 1/4 of your XFS file system size so the tests can pass.

4.3 First Contribution to Swift

4.3.1 Getting Swift

Swifts source code is hosted on github and managed with git. The current trunk can be checked out like this:

```
git clone https://github.com/openstack/swift.git
```

This will clone the Swift repository under your account.

A source tarball for the latest release of Swift is available on the launchpad project page.

Prebuilt packages for Ubuntu and RHEL variants are available.

- Swift Ubuntu Packages
- Swift RDO Packages

4.3.2 Source Control Setup

Swift uses `git` for source control. The OpenStack Developers Guide describes the steps for setting up Git and all the necessary accounts for contributing code to Swift.

4.3.3 Changes to Swift

Once you have the source code and source control set up, you can make your changes to Swift.

4.3.4 Testing

The *Development Guidelines* describe the testing requirements before submitting Swift code.

In summary, you can execute tox from the swift home directory (where you checked out the source code):

```
tox
```

Tox will present tests results. Notice that in the beginning, it is very common to break many coding style guidelines.

4.3.5 Proposing changes to Swift

The OpenStack Developers Guide describes the most common `git` commands that you will need.

Following is a list of the commands that you need to know for your first contribution to Swift:

To clone a copy of Swift:

```
git clone https://github.com/openstack/swift.git
```

Under the swift directory, set up the Gerrit repository. The following command configures the repository to know about Gerrit and installs the `Change-Id` commit hook. You only need to do this once:

```
git review -s
```

To create your development branch (substitute branch_name for a name of your choice:

```
git checkout -b <branch_name>
```

To check the files that have been updated in your branch:

```
git status
```

To check the differences between your branch and the repository:

```
git diff
```

Assuming you have not added new files, you commit all your changes using:

```
git commit -a
```

Read the Summary of Git commit message structure for best practices on writing the commit message. When you are ready to send your changes for review use:

```
git review
```

If successful, Git response message will contain a URL you can use to track your changes.

If you need to make further changes to the same review, you can commit them using:

```
git commit -a --amend
```

This will commit the changes under the same set of changes you issued earlier. Notice that in order to send your latest version for review, you will still need to call:

```
git review
```

4.3.6 Tracking your changes

After proposing changes to Swift, you can track them at https://review.opendev.org. After logging in, you will see a dashboard of Outgoing reviews for changes you have proposed, Incoming reviews for changes you are reviewing, and Recently closed changes for which you were either a reviewer or owner.

4.3.7 Post rebase instructions

After rebasing, the following steps should be performed to rebuild the swift installation. Note that these commands should be performed from the root of the swift repo directory (e.g. `$HOME/swift/`):

```
sudo python setup.py develop
sudo pip install -r test-requirements.txt
```

If using TOX, depending on the changes made during the rebase, you may need to rebuild the TOX environment (generally this will be the case if test-requirements.txt was updated such that a new version of a package is required), this can be accomplished using the `-r` argument to the TOX cli:

```
tox -r
```

You can include any of the other TOX arguments as well, for example, to run the pep8 suite and rebuild the TOX environment the following can be used:

```
tox -r -e pep8
```

The rebuild option only needs to be specified once for a particular build (e.g. pep8), that is further invocations of the same build will not require this until the next rebase.

4.3.8 Troubleshooting

You may run into the following errors when starting Swift if you rebase your commit using:

```
git rebase
```

```
Traceback (most recent call last):
    File "/usr/local/bin/swift-init", line 5, in <module>
        from pkg_resources import require
    File "/usr/lib/python2.7/dist-packages/pkg_resources.py", line 2749, in
↪<module>
        working_set = WorkingSet._build_master()
    File "/usr/lib/python2.7/dist-packages/pkg_resources.py", line 446, in _
↪build_master
        return cls._build_from_requirements(__requires__)
    File "/usr/lib/python2.7/dist-packages/pkg_resources.py", line 459, in _
↪build_from_requirements
        dists = ws.resolve(reqs, Environment())
```

(continues on next page)

(continued from previous page)

```
    File "/usr/lib/python2.7/dist-packages/pkg_resources.py", line 628, in␣
↪resolve
        raise DistributionNotFound(req)
pkg_resources.DistributionNotFound: swift==2.3.1.devXXX
```

(where XXX represents a dev version of Swift).

```
Traceback (most recent call last):
    File "/usr/local/bin/swift-proxy-server", line 10, in <module>
      execfile(__file__)
    File "/home/swift/swift/bin/swift-proxy-server", line 23, in <module>
      sys.exit(run_wsgi(conf_file, 'proxy-server', **options))
    File "/home/swift/swift/swift/common/wsgi.py", line 888, in run_wsgi
      loadapp(conf_path, global_conf=global_conf)
    File "/home/swift/swift/swift/common/wsgi.py", line 390, in loadapp
      func(PipelineWrapper(ctx))
    File "/home/swift/swift/swift/proxy/server.py", line 602, in modify_wsgi_
↪pipeline
      ctx = pipe.create_filter(filter_name)
    File "/home/swift/swift/swift/common/wsgi.py", line 329, in create_filter
      global_conf=self.context.global_conf)
    File "/usr/lib/python2.7/dist-packages/paste/deploy/loadwsgi.py", line␣
↪296, in loadcontext
      global_conf=global_conf)
    File "/usr/lib/python2.7/dist-packages/paste/deploy/loadwsgi.py", line␣
↪328, in _loadegg
      return loader.get_context(object_type, name, global_conf)
    File "/usr/lib/python2.7/dist-packages/paste/deploy/loadwsgi.py", line␣
↪620, in get_context
      object_type, name=name)
    File "/usr/lib/python2.7/dist-packages/paste/deploy/loadwsgi.py", line␣
↪659, in find_egg_entry_point
      for prot in protocol_options] or '(no entry points)'))))
LookupError: Entry point 'versioned_writes' not found in egg 'swift' (dir: /
↪home/swift/swift; protocols: paste.filter_factory, paste.filter_app_factory;
↪ entry_points: )
```

This happens because `git rebase` will retrieve code for a different version of Swift in the development stream, but the start scripts under `/usr/local/bin` have not been updated. The solution is to follow the steps described in the *Post rebase instructions* section.

4.4 Adding Storage Policies to an Existing SAIO

Depending on when you downloaded your SAIO environment, it may already be prepared with two storage policies that enable some basic functional tests. In the event that you are adding a storage policy to an existing installation, however, the following section will walk you through the steps for setting up Storage Policies. Note that configuring more than one storage policy on your development environment is recommended but optional. Enabling multiple Storage Policies is very easy regardless of whether you are working with an existing installation or starting a brand new one.

Now we will create two policies - the first one will be a standard triple replication policy that we will also explicitly set as the default and the second will be setup for reduced replication using a factor of 2x. We will call the first one gold and the second one silver. In this example both policies map to the same devices because its also important for this sample implementation to be simple and easy to understand and adding a bunch of new devices isnt really required to implement a usable set of policies.

1. To define your policies, add the following to your `/etc/swift/swift.conf` file:

```
[storage-policy:0]
name = gold
aliases = yellow, orange
default = yes

[storage-policy:1]
name = silver
```

   See *Storage Policies* for detailed information on `swift.conf` policy options.

2. To create the object ring for the silver policy (index 1), add the following to your `bin/remakerings` script and re-run it (your script may already have these changes):

```
swift-ring-builder object-1.builder create 10 2 1
swift-ring-builder object-1.builder add r1z1-127.0.0.1:6210/sdb1 1
swift-ring-builder object-1.builder add r1z2-127.0.0.1:6220/sdb2 1
swift-ring-builder object-1.builder add r1z3-127.0.0.1:6230/sdb3 1
swift-ring-builder object-1.builder add r1z4-127.0.0.1:6240/sdb4 1
swift-ring-builder object-1.builder rebalance
```

   Note that the reduced replication of the silver policy is only a function of the replication parameter in the `swift-ring-builder create` command and is not specified in `/etc/swift/swift.conf`.

3. Copy `etc/container-reconciler.conf-sample` to `/etc/swift/container-reconciler.conf` and fix the user option:

```
cp etc/container-reconciler.conf-sample /etc/swift/container-reconciler.
↪conf
sed -i "s/# user.*/user = $USER/g" /etc/swift/container-reconciler.conf
```

4.4.1 Using Policies

Setting up Storage Policies was very simple, and using them is even simpler. In this section, we will run some commands to create a few containers with different policies and store objects in them and see how Storage Policies effect placement of data in Swift.

1. We will be using the list_endpoints middleware to confirm object locations, so enable that now in your `proxy-server.conf` file by adding it to the pipeline and including the filter section as shown below (be sure to restart your proxy after making these changes):

```
pipeline = catch_errors gatekeeper healthcheck proxy-logging cache bulk \
  slo dlo ratelimit crossdomain list-endpoints tempurl tempauth staticweb␣
↪\
  container-quotas account-quotas proxy-logging proxy-server

[filter:list-endpoints]
use = egg:swift#list_endpoints
```

2. Check to see that your policies are reported via /info:

```
swift -A http://127.0.0.1:8080/auth/v1.0 -U test:tester -K testing info
```

   You should see this: (only showing the policy output here):

```
policies: [{'aliases': 'gold, yellow, orange', 'default': True,
    'name': 'gold'}, {'aliases': 'silver', 'name': 'silver'}]
```

3. Now create a container without specifying a policy, it will use the default, gold and then put a test object in it (create the file `file0.txt` with your favorite editor with some content):

```
curl -v -X PUT -H 'X-Auth-Token: <your auth token>' \
    http://127.0.0.1:8080/v1/AUTH_test/myCont0
curl -X PUT -v -T file0.txt -H 'X-Auth-Token: <your auth token>' \
    http://127.0.0.1:8080/v1/AUTH_test/myCont0/file0.txt
```

4. Now confirm placement of the object with the *List Endpoints* middleware:

```
curl -X GET -v http://127.0.0.1:8080/endpoints/AUTH_test/myCont0/file0.txt
```

   You should see this: (note placement on expected devices):

```
["http://127.0.0.1:6230/sdb3/761/AUTH_test/myCont0/file0.txt",
 "http://127.0.0.1:6210/sdb1/761/AUTH_test/myCont0/file0.txt",
 "http://127.0.0.1:6220/sdb2/761/AUTH_test/myCont0/file0.txt"]
```

5. Create a container using policy silver and put a different file in it:

```
curl -v -X PUT -H 'X-Auth-Token: <your auth token>' -H \
    "X-Storage-Policy: silver" \
    http://127.0.0.1:8080/v1/AUTH_test/myCont1
curl -X PUT -v -T file1.txt -H 'X-Auth-Token: <your auth token>' \
    http://127.0.0.1:8080/v1/AUTH_test/myCont1/
```

6. Confirm placement of the object for policy silver:

```
curl -X GET -v http://127.0.0.1:8080/endpoints/AUTH_test/myCont1/file1.txt
```

You should see this: (note placement on expected devices):

```
["http://127.0.0.1:6210/sdb1/32/AUTH_test/myCont1/file1.txt",
 "http://127.0.0.1:6240/sdb4/32/AUTH_test/myCont1/file1.txt"]
```

7. Confirm account information with HEAD, make sure that your container-updater service is running and has executed once since you performed the PUTs or the account database wont be updated yet:

```
curl -i -X HEAD -H 'X-Auth-Token: <your auth token>' \
    http://127.0.0.1:8080/v1/AUTH_test
```

You should see something like this (note that total and per policy stats object sizes will vary):

```
HTTP/1.1 204 No Content
Content-Length: 0
X-Account-Object-Count: 2
X-Account-Bytes-Used: 174
X-Account-Container-Count: 2
X-Account-Storage-Policy-Gold-Object-Count: 1
X-Account-Storage-Policy-Gold-Bytes-Used: 84
X-Account-Storage-Policy-Silver-Object-Count: 1
X-Account-Storage-Policy-Silver-Bytes-Used: 90
X-Timestamp: 1397230339.71525
Content-Type: text/plain; charset=utf-8
Accept-Ranges: bytes
X-Trans-Id: tx96e7496b19bb44abb55a3-0053482c75
X-Openstack-Request-Id: tx96e7496b19bb44abb55a3-0053482c75
Date: Fri, 11 Apr 2014 17:55:01 GMT
```

4.5 Auth Server and Middleware

4.5.1 Creating Your Own Auth Server and Middleware

The included swift/common/middleware/tempauth.py is a good example of how to create an auth subsystem with proxy server auth middleware. The main points are that the auth middleware can reject requests up front, before they ever get to the Swift Proxy application, and afterwards when the proxy issues callbacks to verify authorization.

Its generally good to separate the authentication and authorization procedures. Authentication verifies that a request actually comes from who it says it does. Authorization verifies the who has access to the resource(s) the request wants.

Authentication is performed on the request before it ever gets to the Swift Proxy application. The identity information is gleaned from the request, validated in some way, and the validation information is added to the WSGI environment as needed by the future authorization procedure. What exactly is added to the WSGI environment is solely dependent on what the installed authorization procedures need; the Swift Proxy application itself needs no specific information, it just passes it along. Convention has environ[REMOTE_USER] set to the authenticated user string but often more information is needed than just that.

The included TempAuth will set the REMOTE_USER to a comma separated list of groups the user belongs to. The first group will be the users group, a group that only the user belongs to. The second group will be the accounts group, a group that includes all users for that auth account (different than the storage account). The third group is optional and is the storage account string. If the user does not have admin access to the account, the third group will be omitted.

It is highly recommended that authentication server implementers prefix their tokens and Swift storage accounts they create with a configurable reseller prefix (`AUTH_` by default with the included TempAuth). This prefix will avoid conflicts with other authentication servers that might be using the same Swift cluster. Otherwise, the Swift cluster will have to try all the resellers until one validates a token or all fail.

A restriction with group names is that no group name should begin with a period . as that is reserved for internal Swift use (such as the .r for referrer designations as youll see later).

Example Authentication with TempAuth:

- Token AUTH_tkabcd is given to the TempAuth middleware in a requests X-Auth-Token header.
- The TempAuth middleware validates the token AUTH_tkabcd and discovers it matches the tester user within the test account for the storage account AUTH_storage_xyz.
- The TempAuth middleware sets the REMOTE_USER to test:tester,test,AUTH_storage_xyz
- Now this user will have full access (via authorization procedures later) to the AUTH_storage_xyz Swift storage account and access to containers in other storage accounts, provided the storage account begins with the same `AUTH_` reseller prefix and the container has an ACL specifying at least one of those three groups.

Authorization is performed through callbacks by the Swift Proxy server to the WSGI environments swift.authorize value, if one is set. The swift.authorize value should simply be a function that takes a Request as an argument and returns None if access is granted or returns a callable(environ, start_response) if access is denied. This callable is a standard WSGI callable. Generally, you should return 403 Forbidden for requests by an authenticated user and 401 Unauthorized for an unauthenticated request. For example, heres an authorize function that only allows GETs (in this case youd probably return 405 Method Not Allowed, but ignore that for the moment).:

```
from swift.common.swob import HTTPForbidden, HTTPUnauthorized


def authorize(req):
    if req.method == 'GET':
        return None
    if req.remote_user:
        return HTTPForbidden(request=req)
    else:
        return HTTPUnauthorized(request=req)
```

Adding the swift.authorize callback is often done by the authentication middleware as authentication and authorization are often paired together. But, you could create separate authorization middleware that simply sets the callback before passing on the request. To continue our example above:

```
from swift.common.swob import HTTPForbidden, HTTPUnauthorized


class Authorization(object):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, app, conf):
        self.app = app
        self.conf = conf

    def __call__(self, environ, start_response):
        environ['swift.authorize'] = self.authorize
        return self.app(environ, start_response)

    def authorize(self, req):
        if req.method == 'GET':
            return None
        if req.remote_user:
            return HTTPForbidden(request=req)
        else:
            return HTTPUnauthorized(request=req)


def filter_factory(global_conf, **local_conf):
    conf = global_conf.copy()
    conf.update(local_conf)
    def auth_filter(app):
        return Authorization(app, conf)
    return auth_filter
```

The Swift Proxy server will call swift.authorize after some initial work, but before truly trying to process the request. Positive authorization at this point will cause the request to be fully processed immediately. A denial at this point will immediately send the denial response for most operations.

But for some operations that might be approved with more information, the additional information will be gathered and added to the WSGI environment and then swift.authorize will be called once more. These are called delay_denial requests and currently include container read requests and object read and write requests. For these requests, the read or write access control string (X-Container-Read and X-Container-Write) will be fetched and set as the acl attribute in the Request passed to swift.authorize.

The delay_denial procedures allow skipping possibly expensive access control string retrievals for requests that can be approved without that information, such as administrator or account owner requests.

To further our example, we now will approve all requests that have the access control string set to same value as the authenticated user string. Note that you probably wouldnt do this exactly as the access control string represents a list rather than a single user, but itll suffice for this example:

```
from swift.common.swob import HTTPForbidden, HTTPUnauthorized


class Authorization(object):

    def __init__(self, app, conf):
        self.app = app
        self.conf = conf
```

(continues on next page)

(continued from previous page)

```
    def __call__(self, environ, start_response):
        environ['swift.authorize'] = self.authorize
        return self.app(environ, start_response)

    def authorize(self, req):
        # Allow anyone to perform GET requests
        if req.method == 'GET':
            return None
        # Allow any request where the acl equals the authenticated user
        if getattr(req, 'acl', None) == req.remote_user:
            return None
        if req.remote_user:
            return HTTPForbidden(request=req)
        else:
            return HTTPUnauthorized(request=req)


def filter_factory(global_conf, **local_conf):
    conf = global_conf.copy()
    conf.update(local_conf)
    def auth_filter(app):
        return Authorization(app, conf)
    return auth_filter
```

The access control string has a standard format included with Swift, though this can be overridden if desired. The standard format can be parsed with swift.common.middleware.acl.parse_acl which converts the string into two arrays of strings: (referrers, groups). The referrers allow comparing the requests Referer header to control access. The groups allow comparing the request.remote_user (or other sources of group information) to control access. Checking referrer access can be accomplished by using the swift.common.middleware.acl.referrer_allowed function. Checking group access is usually a simple string comparison.

Lets continue our example to use parse_acl and referrer_allowed. Now well only allow GETs after a referrer check and any requests after a group check:

```
from swift.common.middleware.acl import parse_acl, referrer_allowed
from swift.common.swob import HTTPForbidden, HTTPUnauthorized


class Authorization(object):

    def __init__(self, app, conf):
        self.app = app
        self.conf = conf

    def __call__(self, environ, start_response):
        environ['swift.authorize'] = self.authorize
        return self.app(environ, start_response)

    def authorize(self, req):
```

(continues on next page)

(continued from previous page)

```
        if hasattr(req, 'acl'):
            referrers, groups = parse_acl(req.acl)
            if req.method == 'GET' and referrer_allowed(req, referrers):
                return None
            if req.remote_user and groups and req.remote_user in groups:
                return None
        if req.remote_user:
            return HTTPForbidden(request=req)
        else:
            return HTTPUnauthorized(request=req)


def filter_factory(global_conf, **local_conf):
    conf = global_conf.copy()
    conf.update(local_conf)
    def auth_filter(app):
        return Authorization(app, conf)
    return auth_filter
```

The access control strings are set with PUTs and POSTs to containers with the X-Container-Read and X-Container-Write headers. Swift allows these strings to be set to any value, though its very useful to validate that the strings meet the desired format and return a useful error to the user if they dont.

To support this validation, the Swift Proxy application will call the WSGI environments swift.clean_acl callback whenever one of these headers is to be written. The callback should take a header name and value as its arguments. It should return the cleaned value to save if valid or raise a ValueError with a reasonable error message if not.

There is an included swift.common.middleware.acl.clean_acl that validates the standard Swift format. Lets improve our example by making use of that:

```
from swift.common.middleware.acl import \
    clean_acl, parse_acl, referrer_allowed
from swift.common.swob import HTTPForbidden, HTTPUnauthorized


class Authorization(object):

    def __init__(self, app, conf):
        self.app = app
        self.conf = conf

    def __call__(self, environ, start_response):
        environ['swift.authorize'] = self.authorize
        environ['swift.clean_acl'] = clean_acl
        return self.app(environ, start_response)

    def authorize(self, req):
        if hasattr(req, 'acl'):
            referrers, groups = parse_acl(req.acl)
```

(continues on next page)

(continued from previous page)

```
            if req.method == 'GET' and referrer_allowed(req, referrers):
                return None
            if req.remote_user and groups and req.remote_user in groups:
                return None
        if req.remote_user:
            return HTTPForbidden(request=req)
        else:
            return HTTPUnauthorized(request=req)


def filter_factory(global_conf, **local_conf):
    conf = global_conf.copy()
    conf.update(local_conf)
    def auth_filter(app):
        return Authorization(app, conf)
    return auth_filter
```

Now, if you want to override the format for access control strings youll have to provide your own clean_acl function and youll have to do your own parsing and authorization checking for that format. Its highly recommended you use the standard format simply to support the widest range of external tools, but sometimes thats less important than meeting certain ACL requirements.

4.5.2 Integrating With repoze.what

Heres an example of integration with repoze.what, though honestly Im no repoze.what expert by any stretch; this is just included here to hopefully give folks a start on their own code if they want to use repoze.what:

```
from time import time

from eventlet.timeout import Timeout
from repoze.what.adapters import BaseSourceAdapter
from repoze.what.middleware import setup_auth
from repoze.what.predicates import in_any_group, NotAuthorizedError
from swift.common.bufferedhttp import http_connect_raw as http_connect
from swift.common.middleware.acl import clean_acl, parse_acl, referrer_allowed
from swift.common.utils import cache_from_env, split_path
from swift.common.swob import HTTPForbidden, HTTPUnauthorized


class DevAuthorization(object):

    def __init__(self, app, conf):
        self.app = app
        self.conf = conf

    def __call__(self, environ, start_response):
        environ['swift.authorize'] = self.authorize
        environ['swift.clean_acl'] = clean_acl
```

(continues on next page)

(continued from previous page)

```
        return self.app(environ, start_response)

    def authorize(self, req):
        version, account, container, obj = split_path(req.path, 1, 4, True)
        if not account:
            return self.denied_response(req)
        referrers, groups = parse_acl(getattr(req, 'acl', None))
        if referrer_allowed(req, referrers):
            return None
        try:
            in_any_group(account, *groups).check_authorization(req.environ)
        except NotAuthorizedError:
            return self.denied_response(req)
        return None

    def denied_response(self, req):
        if req.remote_user:
            return HTTPForbidden(request=req)
        else:
            return HTTPUnauthorized(request=req)


class DevIdentifier(object):

    def __init__(self, conf):
        self.conf = conf

    def identify(self, env):
        return {'token':
                env.get('HTTP_X_AUTH_TOKEN', env.get('HTTP_X_STORAGE_TOKEN'))}

    def remember(self, env, identity):
        return []

    def forget(self, env, identity):
        return []


class DevAuthenticator(object):

    def __init__(self, conf):
        self.conf = conf
        self.auth_host = conf.get('ip', '127.0.0.1')
        self.auth_port = int(conf.get('port', 11000))
        self.ssl = \
            conf.get('ssl', 'false').lower() in ('true', 'on', '1', 'yes')
        self.auth_prefix = conf.get('prefix', '/')
        self.timeout = float(conf.get('node_timeout', 10))
```

(continues on next page)

(continued from previous page)

```
    def authenticate(self, env, identity):
        token = identity.get('token')
        if not token:
            return None
        memcache_client = cache_from_env(env)
        key = 'devauth/%s' % token
        cached_auth_data = memcache_client.get(key)
        if cached_auth_data:
            start, expiration, user = cached_auth_data
            if time() - start <= expiration:
                return user
        with Timeout(self.timeout):
            conn = http_connect(self.auth_host, self.auth_port, 'GET',
                    '%stoken/%s' % (self.auth_prefix, token), ssl=self.ssl)
            resp = conn.getresponse()
            resp.read()
            conn.close()
        if resp.status == 204:
            expiration = float(resp.getheader('x-auth-ttl'))
            user = resp.getheader('x-auth-user')
            memcache_client.set(key, (time(), expiration, user),
                                time=expiration)
            return user
        return None


class DevChallenger(object):

    def __init__(self, conf):
        self.conf = conf

    def challenge(self, env, status, app_headers, forget_headers):
        def no_challenge(env, start_response):
            start_response(str(status), [])
            return []
        return no_challenge


class DevGroupSourceAdapter(BaseSourceAdapter):

    def __init__(self, *args, **kwargs):
        super(DevGroupSourceAdapter, self).__init__(*args, **kwargs)
        self.sections = {}

    def _get_all_sections(self):
        return self.sections

    def _get_section_items(self, section):
        return self.sections[section]
```

(continues on next page)

(continued from previous page)

```
    def _find_sections(self, credentials):
        return credentials['repoze.what.userid'].split(',')

    def _include_items(self, section, items):
        self.sections[section] |= items

    def _exclude_items(self, section, items):
        for item in items:
            self.sections[section].remove(item)

    def _item_is_included(self, section, item):
        return item in self.sections[section]

    def _create_section(self, section):
        self.sections[section] = set()

    def _edit_section(self, section, new_section):
        self.sections[new_section] = self.sections[section]
        del self.sections[section]

    def _delete_section(self, section):
        del self.sections[section]

    def _section_exists(self, section):
        return self.sections.has_key(section)


class DevPermissionSourceAdapter(BaseSourceAdapter):

    def __init__(self, *args, **kwargs):
        super(DevPermissionSourceAdapter, self).__init__(*args, **kwargs)
        self.sections = {}

    def _get_all_sections(self):
        return self.sections

    def _get_section_items(self, section):
        return self.sections[section]

    def _find_sections(self, group_name):
        return set([n for (n, p) in self.sections.items()
                    if group_name in p])

    def _include_items(self, section, items):
        self.sections[section] |= items

    def _exclude_items(self, section, items):
        for item in items:
```

(continues on next page)

(continued from previous page)

```
            self.sections[section].remove(item)

    def _item_is_included(self, section, item):
        return item in self.sections[section]

    def _create_section(self, section):
        self.sections[section] = set()

    def _edit_section(self, section, new_section):
        self.sections[new_section] = self.sections[section]
        del self.sections[section]

    def _delete_section(self, section):
        del self.sections[section]

    def _section_exists(self, section):
        return self.sections.has_key(section)


def filter_factory(global_conf, **local_conf):
    conf = global_conf.copy()
    conf.update(local_conf)
    def auth_filter(app):
        return setup_auth(DevAuthorization(app, conf),
            group_adapters={'all_groups': DevGroupSourceAdapter()},
            permission_adapters={'all_perms': DevPermissionSourceAdapter()},
            identifiers=[('devauth', DevIdentifier(conf))],
            authenticators=[('devauth', DevAuthenticator(conf))],
            challengers=[('devauth', DevChallenger(conf))])
    return auth_filter
```

4.5.3 Allowing CORS with Auth

Cross Origin Resource Sharing (CORS) require that the auth system allow the OPTIONS method to pass through without a token. The preflight request will make an OPTIONS call against the object or container and will not work if the auth system stops it. See TempAuth for an example of how OPTIONS requests are handled.

4.6 Middleware and Metadata

4.6.1 Using Middleware

Python WSGI Middleware (or just middleware) can be used to wrap the request and response of a Python WSGI application (i.e. a webapp, or REST/HTTP API), like Swifts WSGI servers (proxy-server, account-server, container-server, object-server). Swift uses middleware to add (sometimes optional) behaviors to the Swift WSGI servers.

Middleware can be added to the Swift WSGI servers by modifying their paste configuration file. The majority of Swift middleware is applied to the *Proxy Server*.

Given the following basic configuration:

```
[DEFAULT]
log_level = DEBUG
user = <your-user-name>

[pipeline:main]
pipeline = proxy-server

[app:proxy-server]
use = egg:swift#proxy
```

You could add the *Healthcheck* middleware by adding a section for that filter and adding it to the pipeline:

```
[DEFAULT]
log_level = DEBUG
user = <your-user-name>

[pipeline:main]
pipeline = healthcheck proxy-server

[filter:healthcheck]
use = egg:swift#healthcheck

[app:proxy-server]
use = egg:swift#proxy
```

Some middleware is required and will be inserted into your pipeline automatically by core swift code (e.g. the proxy-server will insert *CatchErrors* and *GateKeeper* at the start of the pipeline if they are not already present). You can see which features are available on a given Swift endpoint (including middleware) using the *Discoverability* interface.

4.6.2 Creating Your Own Middleware

The best way to see how to write middleware is to look at examples.

Many optional features in Swift are implemented as *Middleware* and provided in `swift.common.middleware`, but Swift middleware may be packaged and distributed as a separate project. Some examples are listed on the *Associated Projects* page.

A contrived middleware example that modifies request behavior by inspecting custom HTTP headers (e.g. X-Webhook) and uses *System Metadata* to persist data to backend storage as well as common patterns like a *`get_container_info()`* cache/query and *`wsgify()`* decorator is presented below:

```
from swift.common.http import is_success
from swift.common.swob import wsgify
from swift.common.utils import split_path, get_logger
from swift.common.request_helpers import get_sys_meta_prefix
from swift.proxy.controllers.base import get_container_info
```

(continues on next page)

(continued from previous page)

```
from eventlet import Timeout
import six
if six.PY3:
    from eventlet.green.urllib import request as urllib2
else:
    from eventlet.green import urllib2

# x-container-sysmeta-webhook
SYSMETA_WEBHOOK = get_sys_meta_prefix('container') + 'webhook'


class WebhookMiddleware(object):
    def __init__(self, app, conf):
        self.app = app
        self.logger = get_logger(conf, log_route='webhook')

    @wsgify
    def __call__(self, req):
        obj = None
        try:
            (version, account, container, obj) = \
                split_path(req.path_info, 4, 4, True)
        except ValueError:
            # not an object request
            pass
        if 'x-webhook' in req.headers:
            # translate user's request header to sysmeta
            req.headers[SYSMETA_WEBHOOK] = \
                req.headers['x-webhook']
        if 'x-remove-webhook' in req.headers:
            # empty value will tombstone sysmeta
            req.headers[SYSMETA_WEBHOOK] = ''
        # account and object storage will ignore x-container-sysmeta-*
        resp = req.get_response(self.app)
        if obj and is_success(resp.status_int) and req.method == 'PUT':
            container_info = get_container_info(req.environ, self.app)
            # container_info may have our new sysmeta key
            webhook = container_info['sysmeta'].get('webhook')
            if webhook:
                # create a POST request with obj name as body
                webhook_req = urllib2.Request(webhook, data=obj)
                with Timeout(20):
                    try:
                        urllib2.urlopen(webhook_req).read()
                    except (Exception, Timeout):
                        self.logger.exception(
                            'failed POST to webhook %s' % webhook)
                    else:
                        self.logger.info(
```

(continues on next page)

(continued from previous page)

```
                                'successfully called webhook %s' % webhook)
        if 'x-container-sysmeta-webhook' in resp.headers:
            # translate sysmeta from the backend resp to
            # user-visible client resp header
            resp.headers['x-webhook'] = resp.headers[SYSMETA_WEBHOOK]
        return resp


def webhook_factory(global_conf, **local_conf):
    conf = global_conf.copy()
    conf.update(local_conf)

    def webhook_filter(app):
        return WebhookMiddleware(app, conf)
    return webhook_filter
```

In practice this middleware will call the URL stored on the container as X-Webhook on all successful object uploads.

If this example was at `<swift-repo>/swift/common/middleware/webhook.py` - you could add it to your proxy by creating a new filter section and adding it to the pipeline:

```
[DEFAULT]
log_level = DEBUG
user = <your-user-name>

[pipeline:main]
pipeline = healthcheck webhook proxy-server

[filter:webhook]
paste.filter_factory = swift.common.middleware.webhook:webhook_factory

[filter:healthcheck]
use = egg:swift#healthcheck

[app:proxy-server]
use = egg:swift#proxy
```

Most python packages expose middleware as entrypoints. See PasteDeploy documentation for more information about the syntax of the `use` option. All middleware included with Swift is installed to support the `egg:swift` syntax.

Middleware may advertize its availability and capabilities via Swifts *Discoverability* support by using *register_swift_info()*:

```
from swift.common.registry import register_swift_info
def webhook_factory(global_conf, **local_conf):
    register_swift_info('webhook')
    def webhook_filter(app):
        return WebhookMiddleware(app)
    return webhook_filter
```

If a middleware handles sensitive information in headers or query parameters that may need redaction when logging, use the *register_sensitive_header()* and *register_sensitive_param()* functions. This should be done in the filter factory:

```
from swift.common.registry import register_sensitive_header
def webhook_factory(global_conf, **local_conf):
    register_sensitive_header('webhook-api-key')
    def webhook_filter(app):
        return WebhookMiddleware(app)
    return webhook_filter
```

4.6.3 Swift Metadata

Generally speaking metadata is information about a resource that is associated with the resource but is not the data contained in the resource itself - which is set and retrieved via HTTP headers. (e.g. the Content-Type of a Swift object that is returned in HTTP response headers)

All user resources in Swift (i.e. account, container, objects) can have user metadata associated with them. Middleware may also persist custom metadata to accounts and containers safely using System Metadata. Some core Swift features which predate sysmeta have added exceptions for custom non-user metadata headers (e.g. *ACLs*, *Large Object Support*)

User Metadata

User metadata takes the form of `X-<type>-Meta-<key>: <value>`, where `<type>` depends on the resources type (i.e. Account, Container, Object) and `<key>` and `<value>` are set by the client.

User metadata should generally be reserved for use by the client or client applications. A perfect example use-case for user metadata is python-swiftclients `X-Object-Meta-Mtime` which it stores on object it uploads to implement its `--changed` option which will only upload files that have changed since the last upload.

New middleware should avoid storing metadata within the User Metadata namespace to avoid potential conflict with existing user metadata when introducing new metadata keys. An example of legacy middleware that borrows the user metadata namespace is *TempURL*. An example of middleware which uses custom non-user metadata to avoid the user metadata namespace is *Static Large Objects*.

User metadata that is stored by a PUT or POST request to a container or account resource persists until it is explicitly removed by a subsequent PUT or POST request that includes a header `X-<type>-Meta-<key>` with no value or a header `X-Remove-<type>-Meta-<key>: <ignored-value>`. In the latter case the `<ignored-value>` is not stored. All user metadata stored with an account or container resource is deleted when the account or container is deleted.

User metadata that is stored with an object resource has a different semantic; object user metadata persists until any subsequent PUT or POST request is made to the same object, at which point all user metadata stored with that object is deleted en-masse and replaced with any user metadata included with the PUT or POST request. As a result, it is not possible to update a subset of the user metadata items stored with an object while leaving some items unchanged.

System Metadata

System metadata takes the form of `X-<type>-Sysmeta-<key>: <value>`, where `<type>` depends on the resources type (i.e. Account, Container, Object) and `<key>` and `<value>` are set by trusted code running in a Swift WSGI Server.

All headers on client requests in the form of `X-<type>-Sysmeta-<key>` will be dropped from the request before being processed by any middleware. All headers on responses from back-end systems in the form of `X-<type>-Sysmeta-<key>` will be removed after all middlewares have processed the response but before the response is sent to the client. See *GateKeeper* middleware for more information.

System metadata provides a means to store potentially private custom metadata with associated Swift resources in a safe and secure fashion without actually having to plumb custom metadata through the core swift servers. The incoming filtering ensures that the namespace can not be modified directly by client requests, and the outgoing filter ensures that removing middleware that uses a specific system metadata key renders it benign. New middleware should take advantage of system metadata.

System metadata may be set on accounts and containers by including headers with a PUT or POST request. Where a header name matches the name of an existing item of system metadata, the value of the existing item will be updated. Otherwise existing items are preserved. A system metadata header with an empty value will cause any existing item with the same name to be deleted.

System metadata may be set on objects using only PUT requests. All items of existing system metadata will be deleted and replaced en-masse by any system metadata headers included with the PUT request. System metadata is neither updated nor deleted by a POST request: updating individual items of system metadata with a POST request is not yet supported in the same way that updating individual items of user metadata is not supported. In cases where middleware needs to store its own metadata with a POST request, it may use Object Transient Sysmeta.

Object Transient-Sysmeta

If middleware needs to store object metadata with a POST request it may do so using headers of the form `X-Object-Transient-Sysmeta-<key>: <value>`.

All headers on client requests in the form of `X-Object-Transient-Sysmeta-<key>` will be dropped from the request before being processed by any middleware. All headers on responses from back-end systems in the form of `X-Object-Transient-Sysmeta-<key>` will be removed after all middlewares have processed the response but before the response is sent to the client. See *GateKeeper* middleware for more information.

Transient-sysmeta updates on an object have the same semantic as user metadata updates on an object (see *User Metadata*) i.e. whenever any PUT or POST request is made to an object, all existing items of transient-sysmeta are deleted en-masse and replaced with any transient-sysmeta included with the PUT or POST request. Transient-sysmeta set by a middleware is therefore prone to deletion by a subsequent client-generated POST request unless the middleware is careful to include its transient-sysmeta with every POST. Likewise, user metadata set by a client is prone to deletion by a subsequent middleware-generated POST request, and for that reason middleware should avoid generating POST requests that are independent of any client request.

Transient-sysmeta deliberately uses a different header prefix to user metadata so that middlewares can avoid potential conflict with user metadata keys.

Transient-sysmeta deliberately uses a different header prefix to system metadata to emphasize the fact that the data is only persisted until a subsequent POST.

4.7 Pluggable On-Disk Back-end APIs

The internal REST API used between the proxy server and the account, container and object server is almost identical to public Swift REST API, but with a few internal extensions (for example, update an account with a new container).

The pluggable back-end APIs for the three REST API servers (account, container, object) abstracts the needs for servicing the various REST APIs from the details of how data is laid out and stored on-disk.

The APIs are documented in the reference implementations for all three servers. For historical reasons, the object server backend reference implementation module is named `diskfile`, while the account and container server backend reference implementation modules are named appropriately.

This API is still under development and not yet finalized.

4.7.1 Back-end API for Account Server REST APIs

Pluggable Back-end for Account Server

class swift.account.backend.**AccountBroker**(*db_file*, *timeout=25*, *logger=None*, *account=None*, *container=None*, *pending_timeout=None*, *stale_reads_ok=False*, *skip_commits=False*)

Encapsulates working with an account database.

create_account_stat_table(*conn*, *put_timestamp*)

Create account_stat table which is specific to the account DB. Not a part of Pluggable Back-ends, internal to the baseline code.

Parameters

- **conn** DB connection object
- **put_timestamp** put timestamp

create_container_table(*conn*)

Create container table which is specific to the account DB.

Parameters **conn** DB connection object

create_policy_stat_table(*conn*)

Create policy_stat table which is specific to the account DB. Not a part of Pluggable Back-ends, internal to the baseline code.

Parameters **conn** DB connection object

empty()

Check if the account DB is empty.

Returns True if the database has no active containers.

get_info()

Get global data for the account.

Returns dict with keys: account, created_at, put_timestamp, delete_timestamp, status_changed_at, container_count, object_count, bytes_used, hash, id

get_policy_stats(*do_migrations=False*)

Get global policy stats for the account.

Parameters **do_migrations** boolean, if True the policy stat dicts will always include the container_count key; otherwise it may be omitted on legacy databases until they are migrated.

Returns dict of policy stats where the key is the policy index and the value is a dictionary like {object_count: M, bytes_used: N, container_count: L}

is_status_deleted()

Only returns true if the status field is set to DELETED.

list_containers_iter(*limit*, *marker*, *end_marker*, *prefix*, *delimiter*, *reverse=False*, *allow_reserved=False*)

Get a list of containers sorted by name starting at marker onward, up to limit entries. Entries will begin with the prefix and will not have the delimiter after the prefix.

Parameters

- **limit** maximum number of entries to get
- **marker** marker query
- **end_marker** end marker query
- **prefix** prefix query
- **delimiter** delimiter for query
- **reverse** reverse the result order.
- **allow_reserved** exclude names with reserved-byte by default

Returns list of tuples of (name, object_count, bytes_used, put_timestamp, 0)

make_tuple_for_pickle(*record*)

Turn this db record dict into the format this service uses for pending pickles.

merge_items(*item_list*, *source=None*)

Merge items into the container table.

Parameters

- **item_list** list of dictionaries of {name, put_timestamp, delete_timestamp, object_count, bytes_used, deleted, storage_policy_index}
- **source** if defined, update incoming_sync with the source

put_container(*name*, *put_timestamp*, *delete_timestamp*, *object_count*, *bytes_used*, *storage_policy_index*)

Create a container with the given attributes.

Parameters

- **name** name of the container to create (a native string)
- **put_timestamp** put_timestamp of the container to create
- **delete_timestamp** delete_timestamp of the container to create
- **object_count** number of objects in the container

- **bytes_used** number of bytes used by the container
- **storage_policy_index** the storage policy for this container

4.7.2 Back-end API for Container Server REST APIs

Pluggable Back-ends for Container Server

class swift.container.backend.**ContainerBroker**(*db_file*, *timeout=25*, *logger=None*, *account=None*, *container=None*, *pending_timeout=None*, *stale_reads_ok=False*, *skip_commits=False*, *force_db_file=False*)

Encapsulates working with a container database.

Note that this may involve multiple on-disk DB files if the container becomes sharded:

- `_db_file` is the path to the legacy container DB name, i.e. `<hash>.db`. This file should exist for an initialised broker that has never been sharded, but will not exist once a container has been sharded.
- *db_files* is a list of existing db files for the broker. This list should have at least one entry for an initialised broker, and should have two entries while a broker is in SHARDING state.
- *db_file* is the path to whichever db is currently authoritative for the container. Depending on the containers state, this may not be the same as the `db_file` argument given to `__init__()`, unless `force_db_file` is True in which case *db_file* is always equal to the `db_file` argument given to `__init__()`.
- `pending_file` is always equal to `_db_file` extended with `.pending`, i.e. `<hash>.db.pending`.

classmethod **create_broker**(*device_path*, *part*, *account*, *container*, *logger=None*, *epoch=None*, *put_timestamp=None*, *storage_policy_index=None*)

Create a ContainerBroker instance. If the db doesnt exist, initialize the db file.

Parameters

- **device_path** device path
- **part** partition number
- **account** account name string
- **container** container name string
- **logger** a logger instance
- **epoch** a timestamp to include in the db filename
- **put_timestamp** initial timestamp if broker needs to be initialized
- **storage_policy_index** the storage policy index

Returns a tuple of (`broker`, `initialized`) where `broker` is an instance of *swift.container.backend.ContainerBroker* and `initialized` is True if the db file was initialized, False otherwise.

create_container_info_table(*conn*, *put_timestamp*, *storage_policy_index*)

Create the container_info table which is specific to the container DB. Not a part of Pluggable Back-ends, internal to the baseline code. Also creates the container_stat view.

Parameters

- **conn** DB connection object
- **put_timestamp** put timestamp
- **storage_policy_index** storage policy index

create_object_table(*conn*)

Create the object table which is specific to the container DB. Not a part of Pluggable Back-ends, internal to the baseline code.

Parameters **conn** DB connection object

create_policy_stat_table(*conn*, *storage_policy_index=0*)

Create policy_stat table.

Parameters

- **conn** DB connection object
- **storage_policy_index** the policy_index the container is being created with

create_shard_range_table(*conn*)

Create the shard_range table which is specific to the container DB.

Parameters **conn** DB connection object

property db_file

Get the path to the primary db file for this broker. This is typically the db file for the most recent sharding epoch. However, if no db files exist on disk, or if `force_db_file` was True when the broker was constructed, then the primary db file is the file passed to the broker constructor.

Returns A path to a db file; the file does not necessarily exist.

property db_files

Gets the cached list of valid db files that exist on disk for this broker.

The cached list may be refreshed by calling *reload_db_files()*.

Returns A list of paths to db files ordered by ascending epoch; the list may be empty.

delete_object(*name*, *timestamp*, *storage_policy_index=0*)

Mark an object deleted.

Parameters

- **name** object name to be deleted
- **timestamp** timestamp when the object was marked as deleted
- **storage_policy_index** the storage policy index for the object

empty()

Check if container DB is empty.

This method uses more stringent checks on object count than `is_deleted()`: this method checks that there are no objects in any policy; if the container is in the process of sharding then both fresh and retiring databases are checked to be empty; if a root container has shard ranges then they are checked to be empty.

Returns True if the database has no active objects, False otherwise

enable_sharding(*epoch*)

Updates this brokers own shard range with the given epoch, sets its state to SHARDING and persists it in the DB.

Parameters **epoch** a `Timestamp`

Returns the brokers updated own shard range.

find_shard_ranges(*shard_size*, *limit=- 1*, *existing_ranges=None*, *minimum_shard_size=1*)

Scans the container db for shard ranges. Scanning will start at the upper bound of the any `existing_ranges` that are given, otherwise at `ShardRange.MIN`. Scanning will stop when `limit` shard ranges have been found or when no more shard ranges can be found. In the latter case, the upper bound of the final shard range will be equal to the upper bound of the container namespace.

This method does not modify the state of the db; callers are responsible for persisting any shard range data in the db.

Parameters

- **shard_size** the size of each shard range
- **limit** the maximum number of shard points to be found; a negative value (default) implies no limit.
- **existing_ranges** an optional list of existing ShardRanges; if given, this list should be sorted in order of upper bounds; the scan for new shard ranges will start at the upper bound of the last existing ShardRange.
- **minimum_shard_size** Minimum size of the final shard range. If this is greater than one then the final shard range may be extended to more than shard_size in order to avoid a further shard range with less minimum_shard_size rows.

Returns a tuple; the first value in the tuple is a list of dicts each having keys {index, lower, upper, object_count} in order of ascending upper; the second value in the tuple is a boolean which is True if the last shard range has been found, False otherwise.

get_all_shard_range_data()

Returns a list of all shard range data, including own shard range and deleted shard ranges.

Returns A list of dict representations of a ShardRange.

get_brokers()

Return a list of brokers for component dbs. The list has two entries while the db state is sharding: the first entry is a broker for the retiring db with `skip_commits` set to `True`; the

second entry is a broker for the fresh db with `skip_commits` set to `False`. For any other db state the list has one entry.

Returns a list of *ContainerBroker*

`get_db_state()`

Returns the current state of on disk db files.

`get_info()`

Get global data for the container.

Returns dict with keys: account, container, created_at, put_timestamp, delete_timestamp, status, status_changed_at, object_count, bytes_used, reported_put_timestamp, reported_delete_timestamp, reported_object_count, reported_bytes_used, hash, id, x_container_sync_point1, x_container_sync_point2, and storage_policy_index, db_state.

`get_info_is_deleted()`

Get the is_deleted status and info for the container.

Returns a tuple, in the form (info, is_deleted) info is a dict as returned by get_info and is_deleted is a boolean.

`get_misplaced_since(start, count)`

Get a list of objects which are in a storage policy different from the containers storage policy.

Parameters

- **start** last reconciler sync point
- **count** maximum number of entries to get

Returns list of dicts with keys: name, created_at, size, content_type, etag, storage_policy_index

`get_objects(limit=None, marker='', end_marker='', include_deleted=None, since_row=None)`

Returns a list of objects, including deleted objects, in all policies. Each object in the list is described by a dict with keys {name, created_at, size, content_type, etag, deleted, storage_policy_index}.

Parameters

- **limit** maximum number of entries to get
- **marker** if set, objects with names less than or equal to this value will not be included in the list.
- **end_marker** if set, objects with names greater than or equal to this value will not be included in the list.
- **include_deleted** if True, include only deleted objects; if False, include only undeleted objects; otherwise (default), include both deleted and undeleted objects.
- **since_row** include only items whose ROWID is greater than the given row id; by default all rows are included.

Returns a list of dicts, each describing an object.

`get_own_shard_range`(*no_default=False*)

Returns a shard range representing this brokers own shard range. If no such range has been persisted in the brokers shard ranges table then a default shard range representing the entire namespace will be returned.

The returned shard range will be updated with the current object stats for this broker and a meta timestamp set to the current time. For these values to be persisted the caller must merge the shard range.

Parameters **no_default** if True and the brokers own shard range is not found in the shard ranges table then None is returned, otherwise a default shard range is returned.

Returns an instance of *ShardRange*

`get_replication_info`()

Get information about the DB required for replication.

Returns dict containing keys from get_info plus max_row and metadata

Note:: get_infos <db_contains_type>_count is translated to just count and metadata is the raw string.

`get_shard_ranges`(*marker=None*, *end_marker=None*, *includes=None*, *reverse=False*, *include_deleted=False*, *states=None*, *include_own=False*, *exclude_others=False*, *fill_gaps=False*)

Returns a list of persisted shard ranges.

Parameters

- **marker** restricts the returned list to shard ranges whose namespace includes or is greater than the marker value.
- **end_marker** restricts the returned list to shard ranges whose namespace includes or is less than the end_marker value.
- **includes** restricts the returned list to the shard range that includes the given value; if `includes` is specified then `marker` and `end_marker` are ignored.
- **reverse** reverse the result order.
- **include_deleted** include items that have the delete marker set
- **states** if specified, restricts the returned list to shard ranges that have the given state(s); can be a list of ints or a single int.
- **include_own** boolean that governs whether the row whose name matches the brokers path is included in the returned list. If True, that row is included, otherwise it is not included. Default is False.
- **exclude_others** boolean that governs whether the rows whose names do not match the brokers path are included in the returned list. If True, those rows are not included, otherwise they are included. Default is False.
- **fill_gaps** if True, insert a modified copy of own shard range to fill any gap between the end of any found shard ranges and the upper bound of own shard range. Gaps enclosed within the found shard ranges are not filled.

Returns a list of instances of *swift.common.utils.ShardRange*

`get_shard_usage`()

Get the aggregate object stats for all shard ranges in states ACTIVE, SHARDING or SHRINKING.

Returns a dict with keys {bytes_used, object_count}

`get_sharding_sysmeta`(*key=None*)

Returns sharding specific info from the brokers metadata.

Parameters **key** if given the value stored under `key` in the sharding info will be returned.

Returns either a dict of sharding info or the value stored under `key` in that dict.

`get_sharding_sysmeta_with_timestamps`()

Returns sharding specific info from the brokers metadata with timestamps.

Parameters **key** if given the value stored under `key` in the sharding info will be returned.

Returns a dict of sharding info with their timestamps.

`is_reclaimable`(*now*, *reclaim_age*)

Check if the broker abstraction is empty, and has been marked deleted for at least a reclaim age.

`is_root_container`()

Returns True if this container is a root container, False otherwise.

A root container is a container that is not a shard of another container.

`list_objects_iter`(*limit*, *marker*, *end_marker*, *prefix*, *delimiter*, *path=None*, *storage_policy_index=0*, *reverse=False*, *include_deleted=False*, *since_row=None*, *transform_func=None*, *all_policies=False*, *allow_reserved=False*)

Get a list of objects sorted by name starting at marker onward, up to limit entries. Entries will begin with the prefix and will not have the delimiter after the prefix.

Parameters

- **limit** maximum number of entries to get
- **marker** marker query
- **end_marker** end marker query
- **prefix** prefix query
- **delimiter** delimiter for query
- **path** if defined, will set the prefix and delimiter based on the path
- **storage_policy_index** storage policy index for query
- **reverse** reverse the result order.
- **include_deleted** if True, include only deleted objects; if False (default), include only undeleted objects; otherwise, include both deleted and undeleted objects.

- **since_row** include only items whose ROWID is greater than the given row id; by default all rows are included.
- **transform_func** an optional function that if given will be called for each object to get a transformed version of the object to include in the listing; should have same signature as `_transform_record()`; defaults to `_transform_record()`.
- **all_policies** if True, include objects for all storage policies ignoring any value given for `storage_policy_index`
- **allow_reserved** exclude names with reserved-byte by default

Returns list of tuples of (name, created_at, size, content_type, etag, deleted)

make_tuple_for_pickle(*record*)

Turn this db record dict into the format this service uses for pending pickles.

merge_items(*item_list*, *source=None*)

Merge items into the object table.

Parameters

- **item_list** list of dictionaries of {name, created_at, size, content_type, etag, deleted, storage_policy_index, ctype_timestamp, meta_timestamp}
- **source** if defined, update incoming_sync with the source

merge_shard_ranges(*shard_ranges*)

Merge shard ranges into the shard range table.

Parameters **shard_ranges** a shard range or a list of shard ranges; each shard range should be an instance of *ShardRange* or a dict representation of a shard range having `SHARD_RANGE_KEYS`.

put_object(*name*, *timestamp*, *size*, *content_type*, *etag*, *deleted=0*, *storage_policy_index=0*, *ctype_timestamp=None*, *meta_timestamp=None*)

Creates an object in the DB with its metadata.

Parameters

- **name** object name to be created
- **timestamp** timestamp of when the object was created
- **size** object size
- **content_type** object content-type
- **etag** object etag
- **deleted** if True, marks the object as deleted and sets the deleted_at timestamp to timestamp
- **storage_policy_index** the storage policy index for the object
- **ctype_timestamp** timestamp of when content_type was last updated
- **meta_timestamp** timestamp of when metadata was last updated

reload_db_files()

Reloads the cached list of valid on disk db files for this broker.

remove_objects(*lower*, *upper*, *max_row=None*)

Removes object records in the given namespace range from the object table.

Note that objects are removed regardless of their storage_policy_index.

Parameters

- **lower** defines the lower bound of object names that will be removed; names greater than this value will be removed; names less than or equal to this value will not be removed.
- **upper** defines the upper bound of object names that will be removed; names less than or equal to this value will be removed; names greater than this value will not be removed. The empty string is interpreted as there being no upper bound.
- **max_row** if specified only rows less than or equal to max_row will be removed

reported(*put_timestamp*, *delete_timestamp*, *object_count*, *bytes_used*)

Update reported stats, available with containers *get_info*.

Parameters

- **put_timestamp** put_timestamp to update
- **delete_timestamp** delete_timestamp to update
- **object_count** object_count to update
- **bytes_used** bytes_used to update

classmethod resolve_shard_range_states(*states*)

Given a list of values each of which may be the name of a state, the number of a state, or an alias, return the set of state numbers described by the list.

The following alias values are supported: listing maps to all states that are considered valid when listing objects; updating maps to all states that are considered valid for redirecting an object update; auditing maps to all states that are considered valid for a shard container that is updating its own shard range table from a root (this currently maps to all states except FOUND).

Parameters **states** a list of values each of which may be the name of a state, the number of a state, or an alias

Returns a set of integer state numbers, or None if no states are given

Raises **ValueError** if any value in the given list is neither a valid state nor a valid alias

set_sharded_state()

Unlinks the brokers retiring DB file.

Returns True if the retiring DB was successfully unlinked, False otherwise.

set_sharding_state()

Creates and initializes a fresh DB file in preparation for sharding a retiring DB. The brokers own shard range must have an epoch timestamp for this method to succeed.

Returns True if the fresh DB was successfully created, False otherwise.

set_sharding_sysmeta(*key*, *value*)

Updates the brokers metadata stored under the given key prefixed with a sharding specific namespace.

Parameters

- **key** metadata key in the sharding metadata namespace.
- **value** metadata value

set_storage_policy_index(*policy_index*, *timestamp=None*)

Update the container_stat policy_index and status_changed_at.

sharding_initiated()

Returns True if a broker has shard range state that would be necessary for sharding to have been initiated, False otherwise.

sharding_required()

Returns True if a broker has shard range state that would be necessary for sharding to have been initiated but has not yet completed sharding, False otherwise.

swift.container.backend.**merge_shards**(*shard_data*, *existing*)

Compares `shard_data` with `existing` and updates `shard_data` with any items of `existing` that take precedence over the corresponding item in `shard_data`.

Parameters

- **shard_data** a dict representation of shard range that may be modified by this method.
- **existing** a dict representation of shard range.

Returns True if `shard data` has any item(s) that are considered to take precedence over the corresponding item in `existing`

swift.container.backend.**update_new_item_from_existing**(*new_item*, *existing*)

Compare the data and meta related timestamps of a new object item with the timestamps of an existing object record, and update the new item with data and/or meta related attributes from the existing record if their timestamps are newer.

The multiple timestamps are encoded into a single string for storing in the created_at column of the objects db table.

Parameters

- **new_item** A dict of object update attributes
- **existing** A dict of existing object attributes

Returns True if any attributes of the new item dict were found to be newer than the existing and therefore not updated, otherwise False implying that the updated item is equal to the existing.

4.7.3 Back-end API for Object Server REST APIs

Disk File Interface for the Swift Object Server

The *DiskFile*, *DiskFileWriter* and *DiskFileReader* classes combined define the on-disk abstraction layer for supporting the object server REST API interfaces (excluding *REPLICATE*). Other implementations wishing to provide an alternative backend for the object server must implement the three classes. An example alternative implementation can be found in the *mem_server.py* and *mem_diskfile.py* modules along size this one.

The *DiskFileManager* is a reference implemenation specific class and is not part of the backend API.

The remaining methods in this module are considered implementation specific and are also not considered part of the backend API.

class swift.obj.diskfile.**AuditLocation**(*path*, *device*, *partition*, *policy*)

Represents an object location to be audited.

Other than being a bucket of data, the only useful thing this does is stringify to a filesystem path so the auditors logs look okay.

class swift.obj.diskfile.**BaseDiskFile**(*mgr*, *device_path*, *partition*, *account=None*, *container=None*, *obj=None*, *_datadir=None*, *policy=None*, *use_splice=False*, *pipe_size=None*, *open_expired=False*, *next_part_power=None*, ***kwargs*)

Manage object files.

This specific implementation manages object files on a disk formatted with a POSIX-compliant file system that supports extended attributes as metadata on a file or directory.

Note: The arguments to the constructor are considered implementation specific. The API does not define the constructor arguments.

The following path format is used for data file locations: <devices_path/<device_dir>/<datadir>/<partdir>/<suffixdir>/<hashdir>/ <datafile>.<ext>

Parameters

- **mgr** associated DiskFileManager instance
- **device_path** path to the target device or drive
- **partition** partition on the device in which the object lives
- **account** account name for the object
- **container** container name for the object
- **obj** object name for the object
- **_datadir** override the full datadir otherwise constructed here
- **policy** the StoragePolicy instance
- **use_splice** if true, use zero-copy splice() to send data
- **pipe_size** size of pipe buffer used in zero-copy operations

- **open_expired** if True, open() will not raise a DiskFileExpired if object is expired
- **next_part_power** the next partition power to be used

create(*size=None*)

Context manager to create a file. We create a temporary file first, and then return a DiskFileWriter object to encapsulate the state.

Note: An implementation is not required to perform on-disk preallocations even if the parameter is specified. But if it does and it fails, it must raise a *DiskFileNoSpace* exception.

Parameters **size** optional initial size of file to explicitly allocate on disk

Raises *DiskFileNoSpace* if a size is specified and allocation fails

delete(*timestamp*)

Delete the object.

This implementation creates a tombstone file using the given timestamp, and removes any older versions of the object file. Any file that has an older timestamp than timestamp will be deleted.

Note: An implementation is free to use or ignore the timestamp parameter.

Parameters **timestamp** timestamp to compare with each file

Raises *DiskFileError* this implementation will raise the same errors as the *create()* method.

property durable_timestamp

Provides the timestamp of the newest data file found in the object directory.

Returns A Timestamp instance, or None if no data file was found.

Raises *DiskFileNotOpen* if the open() method has not been previously called on this instance.

get_datafile_metadata()

Provide the datafile metadata for a previously opened object as a dictionary. This is metadata that was included when the object was first PUT, and does not include metadata set by any subsequent POST.

Returns objects datafile metadata dictionary

Raises *DiskFileNotOpen* if the `swift.obj.diskfile.DiskFile.open()` method was not previously invoked

get_metadata()

Provide the metadata for a previously opened object as a dictionary.

Returns objects metadata dictionary

Raises *DiskFileNotOpen* if the `swift.obj.diskfile.DiskFile.open()` method was not previously invoked

`get_metafile_metadata`()

Provide the metafile metadata for a previously opened object as a dictionary. This is metadata that was written by a POST and does not include any persistent metadata that was set by the original PUT.

Returns objects .meta file metadata dictionary, or None if there is no .meta file

Raises *DiskFileNotOpen* if the `swift.obj.diskfile.DiskFile.open()` method was not previously invoked

`open`(*modernize=False*, *current_time=None*)

Open the object.

This implementation opens the data file representing the object, reads the associated metadata in the extended attributes, additionally combining metadata from fast-POST *.meta* files.

Parameters

- **modernize** if set, update this diskfile to the latest format. Currently, this means adding metadata checksums if none are present.
- **current_time** Unix time used in checking expiration. If not present, the current time will be used.

Note: An implementation is allowed to raise any of the following exceptions, but is only required to raise *DiskFileNotExist* when the object representation does not exist.

Raises

- *DiskFileCollision* on name mis-match with metadata
- *DiskFileNotExist* if the object does not exist
- *DiskFileDeleted* if the object was previously deleted
- *DiskFileQuarantined* if while reading metadata of the file some data did pass cross checks

Returns itself for use as a context manager

`read_metadata`(*current_time=None*)

Return the metadata for an object without requiring the caller to open the object first.

Parameters **current_time** Unix time used in checking expiration. If not present, the current time will be used.

Returns metadata dictionary for an object

Raises *DiskFileError* this implementation will raise the same errors as the *open()* method.

`reader`(*keep_cache=False*, *_quarantine_hook=<function BaseDiskFile.<lambda>*)

Return a *swift.common.swob.Response* class compatible *app_iter* object as defined by *swift.obj.diskfile.DiskFileReader*.

For this implementation, the responsibility of closing the open file is passed to the *swift.obj.diskfile.DiskFileReader* object.

Parameters

- **keep_cache** callers preference for keeping data read in the OS buffer cache
- **_quarantine_hook** 1-arg callable called when obj quarantined; the arg is the reason for quarantine. Default is to ignore it. Not needed by the REST layer.

Returns a *swift.obj.diskfile.DiskFileReader* object

write_metadata(*metadata*)

Write a block of metadata to an object without requiring the caller to create the object first. Supports fast-POST behavior semantics.

Parameters **metadata** dictionary of metadata to be associated with the object

Raises *DiskFileError* this implementation will raise the same errors as the *create()* method.

class swift.obj.diskfile.**BaseDiskFileManager**(*conf*, *logger*)

Management class for devices, providing common place for shared parameters and methods not provided by the DiskFile class (which primarily services the object server REST API layer).

The *get_diskfile()* method is how this implementation creates a *DiskFile* object.

Note: This class is reference implementation specific and not part of the pluggable on-disk backend API.

Note: TODO(portante): Not sure what the right name to recommend here, as manager seemed generic enough, though suggestions are welcome.

Parameters

- **conf** caller provided configuration object
- **logger** caller provided logger

cleanup_ondisk_files(*hsh_path*, ***kwargs*)

Clean up on-disk files that are obsolete and gather the set of valid on-disk files for an object.

Parameters

- **hsh_path** object hash path
- **frag_index** if set, search for a specific fragment index .data file, otherwise accept the first valid .data file

Returns a dict that may contain: valid on disk files keyed by their filename extension; a list of obsolete files stored under the key obsolete; a list of files remaining in the directory, reverse sorted, stored under the key files.

static consolidate_hashes(*partition_dir*)

Take whats in hashes.pkl and hashes.invalid, combine them, write the result back to hashes.pkl, and clear out hashes.invalid.

Parameters **partition_dir** absolute path to partition dir containing hashes.pkl and hashes.invalid

Returns a dict, the suffix hashes (if any), the key valid will be False if hashes.pkl is corrupt, cannot be read or does not exist

construct_dev_path(*device*)

Construct the path to a device without checking if it is mounted.

Parameters **device** name of target device

Returns full path to the device

get_dev_path(*device*, *mount_check=None*)

Return the path to a device, first checking to see if either it is a proper mount point, or at least a directory depending on the mount_check configuration option.

Parameters

- **device** name of target device
- **mount_check** whether or not to check mountedness of device. Defaults to bool(self.mount_check).

Returns full path to the device, None if the path to the device is not a proper mount point or directory.

get_diskfile(*device*, *partition*, *account*, *container*, *obj*, *policy*, ***kwargs*)

Returns a BaseDiskFile instance for an object based on the objects partition, path parts and policy.

Parameters

- **device** name of target device
- **partition** partition on device in which the object lives
- **account** account name for the object
- **container** container name for the object
- **obj** object name for the object
- **policy** the StoragePolicy instance

get_diskfile_and_filenames_from_hash(*device*, *partition*, *object_hash*, *policy*, ***kwargs*)

Returns a tuple of (a DiskFile instance for an object at the given object_hash, the basenames of the files in the objects hash dir). Just in case someone thinks of refactoring, be sure DiskFileDeleted is *not* raised, but the DiskFile instance representing the tombstoned object is returned instead.

Parameters

- **device** name of target device
- **partition** partition on the device in which the object lives
- **object_hash** the hash of an object path
- **policy** the StoragePolicy instance

Raises *DiskFileNotExist* if the object does not exist

Returns a tuple comprising (an instance of BaseDiskFile, a list of file basenames)

get_diskfile_from_audit_location(*audit_location*)

Returns a BaseDiskFile instance for an object at the given AuditLocation.

Parameters **audit_location** object location to be audited

get_diskfile_from_hash(*device*, *partition*, *object_hash*, *policy*, ***kwargs*)

Returns a DiskFile instance for an object at the given object_hash. Just in case someone thinks of refactoring, be sure DiskFileDeleted is *not* raised, but the DiskFile instance representing the tombstoned object is returned instead.

Parameters

- **device** name of target device
- **partition** partition on the device in which the object lives
- **object_hash** the hash of an object path
- **policy** the StoragePolicy instance

Raises *DiskFileNotExist* if the object does not exist

Returns an instance of BaseDiskFile

get_hashes(*device*, *partition*, *suffixes*, *policy*, *skip_rehash=False*)

Parameters

- **device** name of target device
- **partition** partition name
- **suffixes** a list of suffix directories to be recalculated
- **policy** the StoragePolicy instance
- **skip_rehash** just mark the suffixes dirty; return None

Returns a dictionary that maps suffix directories

get_ondisk_files(*files*, *datadir*, *verify=True*, *policy=None*, ***kwargs*)

Given a simple list of files names, determine the files that constitute a valid fileset i.e. a set of files that defines the state of an object, and determine the files that are obsolete and could be deleted. Note that some files may fall into neither category.

If a file is considered part of a valid fileset then its info dict will be added to the results dict, keyed by <extension>_info. Any files that are no longer required will have their info dicts added to a list stored under the key obsolete.

The results dict will always contain entries with keys ts_file, data_file and meta_file. Their values will be the fully qualified path to a file of the corresponding type if there is such a file in the valid fileset, or None.

Parameters

- **files** a list of file names.
- **datadir** directory name files are from; this is used to construct file paths in the results, but the datadir is not modified by this method.

- **verify** if True verify that the ondisk file contract has not been violated, otherwise do not verify.
- **policy** storage policy used to store the files. Used to validate fragment indexes for EC policies.

Returns

a dict that will contain keys: ts_file -> path to a .ts file or None data_file -> path to a .data file or None meta_file -> path to a .meta file or None ctype_file -> path to a .meta file or None

and may contain keys: ts_info -> a file info dict for a .ts file data_info -> a file info dict for a .data file meta_info -> a file info dict for a .meta file ctype_info -> a file info dict for a .meta file which contains the content-type value unexpected -> a list of file paths for unexpected files possible_reclaim -> a list of file info dicts for possible reclaimable files obsolete -> a list of file info dicts for obsolete files

static invalidate_hash(*suffix_dir*)

Invalidates the hash for a suffix_dir in the partitions hashes file.

Parameters **suffix_dir** absolute path to suffix dir whose hash needs invalidating

make_on_disk_filename(*timestamp*, *ext=None*, *ctype_timestamp=None*, **a*, ***kw*)

Returns filename for given timestamp.

Parameters

- **timestamp** the object timestamp, an instance of *Timestamp*
- **ext** an optional string representing a file extension to be appended to the returned file name
- **ctype_timestamp** an optional content-type timestamp, an instance of *Timestamp*

Returns a file name

object_audit_location_generator(*policy*, *device_dirs=None*, *auditor_type='ALL'*)

Yield an AuditLocation for all objects stored under device_dirs.

Parameters

- **policy** the StoragePolicy instance
- **device_dirs** directory of target device
- **auditor_type** either ALL or ZBF

parse_on_disk_filename(*filename*, *policy*)

Parse an on disk file name.

Parameters

- **filename** the file name including extension
- **policy** storage policy used to store the file

Returns

a dict, with keys for timestamp, ext and ctype_timestamp:

- timestamp is a *Timestamp*
- ctype_timestamp is a *Timestamp* or None for .meta files, otherwise None
- ext is a string, the file extension including the leading dot or the empty string if the filename has no extension.

Subclasses may override this method to add further keys to the returned dict.

Raises *DiskFileError* if any part of the filename is not able to be validated.

partition_lock(*device*, *policy*, *partition*, *name=None*, *timeout=None*)

A context manager that will lock on the partition given.

Parameters

- **device** device targeted by the lock request
- **policy** policy targeted by the lock request
- **partition** partition targeted by the lock request

Raises *PartitionLockTimeout* If the lock on the partition cannot be granted within the configured timeout.

pickle_async_update(*device*, *account*, *container*, *obj*, *data*, *timestamp*, *policy*)

Write data describing a container update notification to a pickle file in the async_pending directory.

Parameters

- **device** name of target device
- **account** account name for the object
- **container** container name for the object
- **obj** object name for the object
- **data** update data to be written to pickle file
- **timestamp** a Timestamp
- **policy** the StoragePolicy instance

static quarantine_renamer(*device_path*, *corrupted_file_path*)

In the case that a file is corrupted, move it to a quarantined area to allow replication to fix it.

Params device_path The path to the device the corrupted file is on.

Params corrupted_file_path The path to the file you want quarantined.

Returns path (str) of directory the file was moved to

Raises **OSError** re-raises non errno.EEXIST / errno.ENOTEMPTY exceptions from rename

replication_lock(*device*, *policy*, *partition*)

A context manager that will lock on the partition and, if configured to do so, on the device given.

Parameters

- **device** name of target device

- **policy** policy targeted by the replication request
- **partition** partition targeted by the replication request

Raises *ReplicationLockTimeout* If the lock on the device cannot be granted within the configured timeout.

yield_hashes(*device*, *partition*, *policy*, *suffixes=None*, ***kwargs*)

Yields tuples of (hash_only, timestamps) for object information stored for the given device, partition, and (optionally) suffixes. If suffixes is None, all stored suffixes will be searched for object hashes. Note that if suffixes is not None but empty, such as [], then nothing will be yielded.

timestamps is a dict which may contain items mapping:

- ts_data -> timestamp of data or tombstone file,
- ts_meta -> timestamp of meta file, if one exists
- **ts_ctype -> timestamp of meta file containing most recent** content-type value, if one exists
- durable -> True if data file at ts_data is durable, False otherwise

where timestamps are instances of *Timestamp*

Parameters

- **device** name of target device
- **partition** partition name
- **policy** the StoragePolicy instance
- **suffixes** optional list of suffix directories to be searched

yield_suffixes(*device*, *partition*, *policy*)

Yields tuples of (full_path, suffix_only) for suffixes stored on the given device and partition.

Parameters

- **device** name of target device
- **partition** partition name
- **policy** the StoragePolicy instance

class swift.obj.diskfile.**BaseDiskFileReader**(*fp*, *data_file*, *obj_size*, *etag*, *disk_chunk_size*, *keep_cache_size*, *device_path*, *logger*, *quarantine_hook*, *use_splice*, *pipe_size*, *diskfile*, *keep_cache=False*)

Encapsulation of the WSGI read context for servicing GET REST API requests. Serves as the context manager object for the *swift.obj.diskfile.DiskFile* classs `swift.obj.diskfile.DiskFile.reader()` method.

Note: The quarantining behavior of this method is considered implementation specific, and is not required of the API.

Note: The arguments to the constructor are considered implementation specific. The API does not define the constructor arguments.

Parameters

- **fp** open file object pointer reference
- **data_file** on-disk data file name for the object
- **obj_size** verified on-disk size of the object
- **etag** expected metadata etag value for entire file
- **disk_chunk_size** size of reads from disk in bytes
- **keep_cache_size** maximum object size that will be kept in cache
- **device_path** on-disk device path, used when quarantining an obj
- **logger** logger caller wants this object to use
- **quarantine_hook** 1-arg callable called w/reason when quarantined
- **use_splice** if true, use zero-copy splice() to send data
- **pipe_size** size of pipe buffer used in zero-copy operations
- **diskfile** the diskfile creating this DiskFileReader instance
- **keep_cache** should resulting reads be kept in the buffer cache

`app_iter_range`(*start*, *stop*)

Returns an iterator over the data file for range (start, stop)

`app_iter_ranges`(*ranges*, *content_type*, *boundary*, *size*)

Returns an iterator over the data file for a set of ranges

`close`()

Close the open file handle if present.

For this specific implementation, this method will handle quarantining the file if necessary.

`zero_copy_send`(*wsockfd*)

Does some magic with splice() and tee() to move stuff from disk to network without ever touching userspace.

Parameters **wsockfd** file descriptor (integer) of the socket out which to send data

`class swift.obj.diskfile.BaseDiskFileWriter`(*name*, *datadir*, *size*, *bytes_per_sync*, *diskfile*, *next_part_power*)

Encapsulation of the write context for servicing PUT REST API requests. Serves as the context manager object for the *swift.obj.diskfile.DiskFile* classs `swift.obj.diskfile.DiskFile.create()` method.

Note: It is the responsibility of the `swift.obj.diskfile.DiskFile.create()` method context manager to close the open file descriptor.

Note: The arguments to the constructor are considered implementation specific. The API does not define the constructor arguments.

Parameters

- **name** name of object from REST API
- **datadir** on-disk directory object will end up in on *swift.obj.diskfile.DiskFileWriter.put()*
- **fd** open file descriptor of temporary file to receive data
- **tmppath** full path name of the opened file descriptor
- **bytes_per_sync** number bytes written between sync calls
- **diskfile** the diskfile creating this DiskFileWriter instance
- **next_part_power** the next partition power to be used

chunks_finished()

Expose internal stats about written chunks.

Returns a tuple, (upload_size, etag)

commit(*timestamp*)

Perform any operations necessary to mark the object as durable. For replication policy type this is a no-op.

Parameters **timestamp** object put timestamp, an instance of *Timestamp*

put(*metadata*)

Finalize writing the file on disk.

Parameters **metadata** dictionary of metadata to be associated with the object

write(*chunk*)

Write a chunk of data to disk. All invocations of this method must come before invoking the :func:

For this implementation, the data is written into a temporary file.

Parameters **chunk** the chunk of data to write as a string object

class swift.obj.diskfile.**DiskFile**(*mgr*, *device_path*, *partition*, *account=None*, *container=None*, *obj=None*, *_datadir=None*, *policy=None*, *use_splice=False*, *pipe_size=None*, *open_expired=False*, *next_part_power=None*, ***kwargs*)

reader_cls

alias of *swift.obj.diskfile.DiskFileReader*

writer_cls

alias of *swift.obj.diskfile.DiskFileWriter*

class swift.obj.diskfile.**DiskFileManager**(*conf*, *logger*)

diskfile_cls

alias of *swift.obj.diskfile.DiskFile*

class swift.obj.diskfile.**DiskFileReader**(*fp*, *data_file*, *obj_size*, *etag*, *disk_chunk_size*, *keep_cache_size*, *device_path*, *logger*, *quarantine_hook*, *use_splice*, *pipe_size*, *diskfile*, *keep_cache=False*)

class swift.obj.diskfile.**DiskFileWriter**(*name*, *datadir*, *size*, *bytes_per_sync*, *diskfile*, *next_part_power*)

put(*metadata*)

Finalize writing the file on disk.

Parameters **metadata** dictionary of metadata to be associated with the object

class swift.obj.diskfile.**ECDiskFile**(**args*, ***kwargs*)

property durable_timestamp

Provides the timestamp of the newest durable file found in the object directory.

Returns A Timestamp instance, or None if no durable file was found.

Raises *DiskFileNotOpen* if the open() method has not been previously called on this instance.

property fragments

Provides information about all fragments that were found in the object directory, including fragments without a matching durable file, and including any fragment chosen to construct the opened diskfile.

Returns A dict mapping <Timestamp instance> -> <list of frag indexes>, or None if the diskfile has not been opened or no fragments were found.

purge(*timestamp*, *frag_index*, *nondurable_purge_delay=0*, *meta_timestamp=None*)

Remove a tombstone file matching the specified timestamp or datafile matching the specified timestamp and fragment index from the object directory.

This provides the EC reconstructor/ssync process with a way to remove a tombstone or fragment from a handoff node after reverting it to its primary node.

The hash will be invalidated, and if empty the hsh_path will be removed immediately.

Parameters

- **timestamp** the object timestamp, an instance of *Timestamp*
- **frag_index** fragment archive index, must be a whole number or None.
- **nondurable_purge_delay** only remove a non-durable data file if its been on disk longer than this many seconds.
- **meta_timestamp** if not None then remove any meta file with this timestamp

reader_cls

alias of *swift.obj.diskfile.ECDiskFileReader*

writer_cls

alias of *swift.obj.diskfile.ECDiskFileWriter*

class swift.obj.diskfile.**ECDiskFileManager**(*conf*, *logger*)

diskfile_cls

alias of *swift.obj.diskfile.ECDiskFile*

make_on_disk_filename(*timestamp*, *ext=None*, *frag_index=None*, *ctype_timestamp=None*, *durable=False*, **a*, ***kw*)

Returns the EC specific filename for given timestamp.

Parameters

- **timestamp** the object timestamp, an instance of *Timestamp*
- **ext** an optional string representing a file extension to be appended to the returned file name
- **frag_index** a fragment archive index, used with .data extension only, must be a whole number.
- **ctype_timestamp** an optional content-type timestamp, an instance of *Timestamp*
- **durable** if True then include a durable marker in data filename.

Returns a file name

Raises *DiskFileError* if ext==.data and the kwarg frag_index is not a whole number

parse_on_disk_filename(*filename*, *policy*)

Returns timestamp(s) and other info extracted from a policy specific file name. For EC policy the data file name includes a fragment index and possibly a durable marker, both of which must be stripped off to retrieve the timestamp.

Parameters **filename** the file name including extension

Returns

a dict, with keys for timestamp, frag_index, durable, ext and ctype_timestamp:

- timestamp is a *Timestamp*
- frag_index is an int or None
- ctype_timestamp is a *Timestamp* or None for .meta files, otherwise None
- ext is a string, the file extension including the leading dot or the empty string if the filename has no extension
- durable is a boolean that is True if the filename is a data file that includes a durable marker

Raises *DiskFileError* if any part of the filename is not able to be validated.

validate_fragment_index(*frag_index*, *policy=None*)

Return int representation of frag_index, or raise a DiskFileError if frag_index is not a whole number.

Parameters

- **frag_index** a fragment archive index
- **policy** storage policy used to validate the index against

class swift.obj.diskfile.**ECDiskFileReader**(*fp*, *data_file*, *obj_size*, *etag*, *disk_chunk_size*, *keep_cache_size*, *device_path*, *logger*, *quarantine_hook*, *use_splice*, *pipe_size*, *diskfile*, *keep_cache=False*)

class swift.obj.diskfile.**ECDiskFileWriter**(*name*, *datadir*, *size*, *bytes_per_sync*, *diskfile*, *next_part_power*)

commit(*timestamp*)

Finalize put by renaming the object data file to include a durable marker. We do this for EC policy because it requires a 2-phase put commit confirmation.

Parameters **timestamp** object put timestamp, an instance of *Timestamp*

Raises *DiskFileError* if the diskfile frag_index has not been set (either during initialisation or a call to put())

put(*metadata*)

The only difference between this method and the replication policy DiskFileWriter method is adding the frag index to the metadata.

Parameters **metadata** dictionary of metadata to be associated with object

swift.obj.diskfile.**consolidate_hashes**(*partition_dir*)

Take whats in hashes.pkl and hashes.invalid, combine them, write the result back to hashes.pkl, and clear out hashes.invalid.

Parameters **partition_dir** absolute path to partition dir containing hashes.pkl and hashes.invalid

Returns a dict, the suffix hashes (if any), the key valid will be False if hashes.pkl is corrupt, cannot be read or does not exist

swift.obj.diskfile.**extract_policy**(*obj_path*)

Extracts the policy for an object (based on the name of the objects directory) given the device-relative path to the object. Returns None in the event that the path is malformed in some way.

The device-relative path is everything after the mount point; for example:

/srv/node/d42/objects-5/30/179/ 485dc017205a81df3af616d917c90179/1401811134.873649.data

would have device-relative path:

objects-5/30/179/485dc017205a81df3af616d917c90179/1401811134.873649.data

Parameters **obj_path** device-relative path of an object, or the full path

Returns a *BaseStoragePolicy* or None

swift.obj.diskfile.**get_async_dir**(*policy_or_index*)

Get the async dir for the given policy.

Parameters **policy_or_index** StoragePolicy instance, or an index (string or int); if None, the legacy Policy-0 is assumed.

Returns async_pending or async_pending-<N> as appropriate

swift.obj.diskfile.**get_data_dir**(*policy_or_index*)

Get the data dir for the given policy.

Parameters **policy_or_index** StoragePolicy instance, or an index (string or int); if None, the legacy Policy-0 is assumed.

Returns objects or objects-<N> as appropriate

swift.obj.diskfile.**get_part_path**(*dev_path*, *policy*, *partition*)

Given the device path, policy, and partition, returns the full path to the partition

swift.obj.diskfile.**get_tmp_dir**(*policy_or_index*)

Get the temp dir for the given policy.

Parameters **policy_or_index** StoragePolicy instance, or an index (string or int); if None, the legacy Policy-0 is assumed.

Returns tmp or tmp-<N> as appropriate

swift.obj.diskfile.**invalidate_hash**(*suffix_dir*)

Invalidates the hash for a suffix_dir in the partitions hashes file.

Parameters **suffix_dir** absolute path to suffix dir whose hash needs invalidating

swift.obj.diskfile.**object_audit_location_generator**(*devices*, *datadir*, *mount_check=True*, *logger=None*, *device_dirs=None*, *auditor_type='ALL'*)

Given a devices path (e.g. /srv/node), yield an AuditLocation for all objects stored under that directory for the given datadir (policy), if device_dirs isnt set. If device_dirs is set, only yield AuditLocation for the objects under the entries in device_dirs. The AuditLocation only knows the path to the hash directory, not to the .data file therein (if any). This is to avoid a double listdir(hash_dir); the DiskFile object will always do one, so we dont.

Parameters

- **devices** parent directory of the devices to be audited
- **datadir** objects directory
- **mount_check** flag to check if a mount check should be performed on devices
- **logger** a logger object
- **device_dirs** a list of directories under devices to traverse
- **auditor_type** either ALL or ZBF

swift.obj.diskfile.**quarantine_renamer**(*device_path*, *corrupted_file_path*)

In the case that a file is corrupted, move it to a quarantined area to allow replication to fix it.

Params device_path The path to the device the corrupted file is on.

Params corrupted_file_path The path to the file you want quarantined.

Returns path (str) of directory the file was moved to

Raises **OSError** re-raises non errno.EEXIST / errno.ENOTEMPTY exceptions from rename

swift.obj.diskfile.**read_hashes**(*partition_dir*)

Read the existing hashes.pkl

Returns a dict, the suffix hashes (if any), the key valid will be False if hashes.pkl is corrupt, cannot be read or does not exist

swift.obj.diskfile.**read_metadata**(*fd*, *add_missing_checksum=False*)

Helper function to read the pickled metadata from an object file.

Parameters

- **fd** file descriptor or filename to load the metadata from
- **add_missing_checksum** if set and checksum is missing, add it

Returns dictionary of metadata

swift.obj.diskfile.**relink_paths**(*target_path*, *new_target_path*, *ignore_missing=True*)

Hard-links a file located in `target_path` using the second path `new_target_path`. Creates intermediate directories if required.

Parameters

- **target_path** current absolute filename
- **new_target_path** new absolute filename for the hardlink
- **ignore_missing** if True then no exception is raised if the link could not be made because `target_path` did not exist, otherwise an OSError will be raised.

Raises OSError if the hard link could not be created, unless the intended hard link already exists or the `target_path` does not exist and `must_exist` if False.

Returns True if the link was created by the call to this method, False otherwise.

swift.obj.diskfile.**write_hashes**(*partition_dir*, *hashes*)

Write hashes to hashes.pkl

The updated key is added to hashes before it is written.

swift.obj.diskfile.**write_metadata**(*fd*, *metadata*, *xattr_size=65536*)

Helper function to write pickled metadata for an object file.

Parameters

- **fd** file descriptor or filename to write the metadata
- **metadata** metadata to write

4.8 Auditor Watchers

4.8.1 Overview

The duty of auditors is to guard Swift against corruption in the storage media. But because auditors crawl all objects, they can be used to program Swift to operate on every object. It is done through an API known as watcher.

Watchers do not have any private view into the cluster. An operator can write a standalone program that walks the directories and performs any desired inspection or maintenance. What watcher brings to the table is a framework to do the same job easily, under resource restrictions already in place for the auditor.

Operations performed by watchers are often site-specific, or else they would be incorporated into Swift already. However, the code in the tree provides a reference implementation for convenience. It is located in swift/obj/watchers/dark_data.py and implements so-called Dark Data Watcher.

Currently, only object auditor supports the watchers.

4.8.2 The API class

The implementation of a watcher is a Python class that may look like this:

```
class MyWatcher(object):

  def __init__(self, conf, logger, **kwargs):
      pass

  def start(self, audit_type, **kwargs):
      pass

  def see_object(self, object_metadata, policy_index, partition,
                 data_file_path, **kwargs):
      pass

  def end(self, **kwargs):
      pass
```

Arguments to watcher methods are passed as keyword arguments, and methods are expected to consume new, unknown arguments.

The method __init__() is used to save configuration and logger at the start of the plug-in.

The method start() is invoked when auditor starts a pass. It usually resets counters. The argument *auditor_type* is string of *ALL* or *ZBF*, according to the type of the auditor running the watcher. Watchers that talk to the network tend to hang off the ALL-type auditor, the lightweight ones are okay with the ZBF-type.

The method end() is the closing bracket for start(). It is typically used to log something, or dump some statistics.

The method see_object() is called when auditor completed an audit of an object. This is where most of the work is done.

The protocol for see_object() allows it to raise a special exception, QuarantienRequested. Auditor catches it and quarantines the object. In general, its okay for watcher methods to throw exceptions, so an author of a watcher plugin does not have to catch them explicitly with a try:; they can be just permitted to bubble up naturally.

4.8.3 Loading the plugins

Swift auditor loads watcher classes from eggs, so it is necessary to wrap the class and provide it an entry point:

```
$ cat /usr/lib/python3.8/site-p*/mywatcher*egg-info/entry_points.txt
[mywatcher.mysection]
mywatcherentry = mywatcher:MyWatcher
```

Operator tells Swift auditor what plugins to load by adding them to object-server.conf in the section [object-auditor]. It is also possible to pass parameters, arriving in the argument conf{ } of method start():

```
[object-auditor]
watchers = mywatcher#mywatcherentry,swift#dark_data

[object-auditor:watcher:mywatcher#mywatcherentry]
myparam=testing2020
```

Do not forget to remove the watcher from auditors when done. Although the API itself is very lightweight, it is common for watchers to incur a significant performance penalty: they can talk to networked services or access additional objects.

4.8.4 Dark Data Watcher

The watcher API is assumed to be under development. Operators who need extensions are welcome to report any needs for more arguments to see_object().

The *Dark Data* watcher has been provided as an example. If an operator wants to create their own watcher, start by copying the provided example template `swift/obj/watchers/dark_data.py` and see if it is sufficient.

CHAPTER

FIVE

ADMINISTRATOR DOCUMENTATION

5.1 Instructions for a Multiple Server Swift Installation

Please refer to the latest official OpenStack Installation Guides for the most up-to-date documentation.

5.1.1 Current Install Guides

- Object Storage installation guide for OpenStack Ocata
- Object Storage installation guide for OpenStack Newton

5.2 Deployment Guide

This document provides general guidance for deploying and configuring Swift. Detailed descriptions of configuration options can be found in the *configuration documentation*.

5.2.1 Hardware Considerations

Swift is designed to run on commodity hardware. RAID on the storage drives is not required and not recommended. Swifts disk usage pattern is the worst case possible for RAID, and performance degrades very quickly using RAID 5 or 6.

5.2.2 Deployment Options

The Swift services run completely autonomously, which provides for a lot of flexibility when architecting the hardware deployment for Swift. The 4 main services are:

1. Proxy Services
2. Object Services
3. Container Services
4. Account Services

The Proxy Services are more CPU and network I/O intensive. If you are using 10g networking to the proxy, or are terminating SSL traffic at the proxy, greater CPU power will be required.

The Object, Container, and Account Services (Storage Services) are more disk and network I/O intensive.

The easiest deployment is to install all services on each server. There is nothing wrong with doing this, as it scales each service out horizontally.

Alternatively, one set of servers may be dedicated to the Proxy Services and a different set of servers dedicated to the Storage Services. This allows faster networking to be configured to the proxy than the storage servers, and keeps load balancing to the proxies more manageable. Storage Services scale out horizontally as storage servers are added, and the overall API throughput can be scaled by adding more proxies.

If you need more throughput to either Account or Container Services, they may each be deployed to their own servers. For example you might use faster (but more expensive) SAS or even SSD drives to get faster disk I/O to the databases.

A high-availability (HA) deployment of Swift requires that multiple proxy servers are deployed and requests are load-balanced between them. Each proxy server instance is stateless and able to respond to requests for the entire cluster.

Load balancing and network design is left as an exercise to the reader, but this is a very important part of the cluster, so time should be spent designing the network for a Swift cluster.

5.2.3 Web Front End Options

Swift comes with an integral web front end. However, it can also be deployed as a request processor of an Apache2 using mod_wsgi as described in *Apache Deployment Guide*.

5.2.4 Preparing the Ring

The first step is to determine the number of partitions that will be in the ring. We recommend that there be a minimum of 100 partitions per drive to insure even distribution across the drives. A good starting point might be to figure out the maximum number of drives the cluster will contain, and then multiply by 100, and then round up to the nearest power of two.

For example, imagine we are building a cluster that will have no more than 5,000 drives. That would mean that we would have a total number of 500,000 partitions, which is pretty close to 2^19, rounded up.

It is also a good idea to keep the number of partitions small (relatively). The more partitions there are, the more work that has to be done by the replicators and other backend jobs and the more memory the rings consume in process. The goal is to find a good balance between small rings and maximum cluster size.

The next step is to determine the number of replicas to store of the data. Currently it is recommended to use 3 (as this is the only value that has been tested). The higher the number, the more storage that is used but the less likely you are to lose data.

It is also important to determine how many zones the cluster should have. It is recommended to start with a minimum of 5 zones. You can start with fewer, but our testing has shown that having at least five zones is optimal when failures occur. We also recommend trying to configure the zones at as high a level as possible to create as much isolation as possible. Some example things to take into consideration can include physical location, power availability, and network connectivity. For example, in a small cluster you might decide to split the zones up by cabinet, with each cabinet having its own power and network connectivity. The zone concept is very abstract, so feel free to use it in whatever way best isolates your data from failure. Each zone exists in a region.

A region is also an abstract concept that may be used to distinguish between geographically separated areas as well as can be used within same datacenter. Regions and zones are referenced by a positive integer.

You can now start building the ring with:

```
swift-ring-builder <builder_file> create <part_power> <replicas> <min_part_
↪hours>
```

This will start the ring build process creating the <builder_file> with 2^<part_power> partitions. <min_part_hours> is the time in hours before a specific partition can be moved in succession (24 is a good value for this).

Devices can be added to the ring with:

```
swift-ring-builder <builder_file> add r<region>z<zone>-<ip>:<port>/<device_
↪name>_<meta> <weight>
```

This will add a device to the ring where <builder_file> is the name of the builder file that was created previously, <region> is the number of the region the zone is in, <zone> is the number of the zone this device is in, <ip> is the ip address of the server the device is in, <port> is the port number that the server is running on, <device_name> is the name of the device on the server (for example: sdb1), <meta> is a string of metadata for the device (optional), and <weight> is a float weight that determines how many partitions are put on the device relative to the rest of the devices in the cluster (a good starting point is 100.0 x TB on the drive).Add each device that will be initially in the cluster.

Once all of the devices are added to the ring, run:

```
swift-ring-builder <builder_file> rebalance
```

This will distribute the partitions across the drives in the ring. It is important whenever making changes to the ring to make all the changes required before running rebalance. This will ensure that the ring stays as balanced as possible, and as few partitions are moved as possible.

The above process should be done to make a ring for each storage service (Account, Container and Object). The builder files will be needed in future changes to the ring, so it is very important that these be kept and backed up. The resulting .tar.gz ring file should be pushed to all of the servers in the cluster. For more information about building rings, running swift-ring-builder with no options will display help text with available commands and options. More information on how the ring works internally can be found in the *Ring Overview*.

5.2.5 Running object-servers Per Disk

The lack of true asynchronous file I/O on Linux leaves the object-server workers vulnerable to misbehaving disks. Because any object-server worker can service a request for any disk, and a slow I/O request blocks the eventlet hub, a single slow disk can impair an entire storage node. This also prevents object servers from fully utilizing all their disks during heavy load.

Another way to get full I/O isolation is to give each disk on a storage node a different port in the storage policy rings. Then set the *servers_per_port* option in the object-server config. NOTE: while the purpose of this config setting is to run one or more object-server worker processes per *disk*, the implementation just runs object-servers per unique port of local devices in the rings. The deployer must combine this option with appropriately-configured rings to benefit from this feature.

Heres an example (abbreviated) old-style ring (2 node cluster with 2 disks each):

```
Devices:    id  region  zone      ip address  port  replication ip␣
↪replication port      name
             0       1     1       1.1.0.1    6200       1.1.0.1          ␣
↪  6200      d1
             1       1     1       1.1.0.1    6200       1.1.0.1          ␣
↪  6200      d2
             2       1     2       1.1.0.2    6200       1.1.0.2          ␣
↪  6200      d3
             3       1     2       1.1.0.2    6200       1.1.0.2          ␣
↪  6200      d4
```

And heres the same ring set up for `servers_per_port`:

```
Devices:    id  region  zone      ip address  port  replication ip␣
↪replication port      name
             0       1     1       1.1.0.1    6200       1.1.0.1          ␣
↪  6200      d1
             1       1     1       1.1.0.1    6201       1.1.0.1          ␣
↪  6201      d2
             2       1     2       1.1.0.2    6200       1.1.0.2          ␣
↪  6200      d3
             3       1     2       1.1.0.2    6201       1.1.0.2          ␣
↪  6201      d4
```

When migrating from normal to `servers_per_port`, perform these steps in order:

1. Upgrade Swift code to a version capable of doing `servers_per_port`.
2. Enable `servers_per_port` with a value greater than zero.
3. Restart `swift-object-server` processes with a SIGHUP. At this point, you will have the `servers_per_port` number of `swift-object-server` processes serving all requests for all disks on each node. This preserves availability, but you should perform the next step as quickly as possible.
4. Push out new rings that actually have different ports per disk on each server. One of the ports in the new ring should be the same as the port used in the old ring (6200 in the example above). This will cover existing proxy-server processes who havent loaded the new ring yet. They can still talk to any storage node regardless of whether or not that storage node has loaded the ring and started object-server processes on the new ports.

If you do not run a separate object-server for replication, then this setting must be available to the object-replicator and object-reconstructor (i.e. appear in the [DEFAULT] config section).

5.2.6 General Service Configuration

Most Swift services fall into two categories. Swifts wsgi servers and background daemons.

For more information specific to the configuration of Swifts wsgi servers with paste deploy see *General Server Configuration*.

Configuration for servers and daemons can be expressed together in the same file for each type of server, or separately. If a required section for the service trying to start is missing there will be an error. The sections not used by the service are ignored.

Consider the example of an object storage node. By convention, configuration for the object-server, object-updater, object-replicator, object-auditor, and object-reconstructor exist in a single file `/etc/swift/object-server.conf`:

```
[DEFAULT]
reclaim_age = 604800

[pipeline:main]
pipeline = object-server

[app:object-server]
use = egg:swift#object

[object-replicator]

[object-updater]

[object-auditor]
```

Swift services expect a configuration path as the first argument:

```
$ swift-object-auditor
Usage: swift-object-auditor CONFIG [options]

Error: missing config path argument
```

If you omit the object-auditor section this file could not be used as the configuration path when starting the `swift-object-auditor` daemon:

```
$ swift-object-auditor /etc/swift/object-server.conf
Unable to find object-auditor config section in /etc/swift/object-server.conf
```

If the configuration path is a directory instead of a file all of the files in the directory with the file extension .conf will be combined to generate the configuration object which is delivered to the Swift service. This is referred to generally as directory based configuration.

Directory based configuration leverages ConfigParsers native multi-file support. Files ending in .conf in the given directory are parsed in lexicographical order. Filenames starting with . are ignored. A mixture of file and directory configuration paths is not supported - if the configuration path is a file only that file will be parsed.

The Swift service management tool `swift-init` has adopted the convention of looking for `/etc/swift/{type}-server.conf.d/` if the file `/etc/swift/{type}-server.conf` file does not exist.

When using directory based configuration, if the same option under the same section appears more than once in different files, the last value parsed is said to override previous occurrences. You can ensure proper override precedence by prefixing the files in the configuration directory with numerical values.:

```
/etc/swift/
    default.base
    object-server.conf.d/
        000_default.conf -> ../default.base
        001_default-override.conf
        010_server.conf
        020_replicator.conf
        030_updater.conf
        040_auditor.conf
```

You can inspect the resulting combined configuration object using the `swift-config` command line tool

5.2.7 General Server Configuration

Swift uses paste.deploy (https://pypi.org/project/Paste/) to manage server configurations. Detailed descriptions of configuration options can be found in the *configuration documentation*.

Default configuration options are set in the `[DEFAULT]` section, and any options specified there can be overridden in any of the other sections BUT ONLY BY USING THE SYNTAX `set option_name = value`. This is the unfortunate way paste.deploy works and Ill try to explain it in full.

First, heres an example paste.deploy configuration file:

```
[DEFAULT]
name1 = globalvalue
name2 = globalvalue
name3 = globalvalue
set name4 = globalvalue

[pipeline:main]
pipeline = myapp

[app:myapp]
use = egg:mypkg#myapp
name2 = localvalue
set name3 = localvalue
set name5 = localvalue
name6 = localvalue
```

The resulting configuration that myapp receives is:

```
global {'__file__': '/etc/mypkg/wsgi.conf', 'here': '/etc/mypkg',
        'name1': 'globalvalue',
        'name2': 'globalvalue',
        'name3': 'localvalue',
        'name4': 'globalvalue',
        'name5': 'localvalue',
```

(continues on next page)

(continued from previous page)

```
        'set name4': 'globalvalue'}
local {'name6': 'localvalue'}
```

So, `name1` got the global value which is fine since its only in the `DEFAULT` section anyway.

`name2` got the global value from `DEFAULT` even though it appears to be overridden in the `app:myapp` subsection. This is just the unfortunate way paste.deploy works (at least at the time of this writing.)

`name3` got the local value from the `app:myapp` subsection because it is using the special paste.deploy syntax of `set option_name = value`. So, if you want a default value for most app/filters but want to override it in one subsection, this is how you do it.

`name4` got the global value from `DEFAULT` since its only in that section anyway. But, since we used the `set` syntax in the `DEFAULT` section even though we shouldnt, notice we also got a `set name4` variable. Weird, but probably not harmful.

`name5` got the local value from the `app:myapp` subsection since its only there anyway, but notice that it is in the global configuration and not the local configuration. This is because we used the `set` syntax to set the value. Again, weird, but not harmful since Swift just treats the two sets of configuration values as one set anyway.

`name6` got the local value from `app:myapp` subsection since its only there, and since we didnt use the `set` syntax, its only in the local configuration and not the global one. Though, as indicated above, there is no special distinction with Swift.

Thats quite an explanation for something that should be so much simpler, but it might be important to know how paste.deploy interprets configuration files. The main rule to remember when working with Swift configuration files is:

Note: Use the `set option_name = value` syntax in subsections if the option is also set in the `[DEFAULT]` section. Dont get in the habit of always using the `set` syntax or youll probably mess up your non-paste.deploy configuration files.

Per policy configuration

Some proxy-server configuration options may be overridden for individual *Storage Policies* by including per-policy config section(s). These options are:

- `sorting_method`
- `read_affinity`
- `write_affinity`
- `write_affinity_node_count`
- `write_affinity_handoff_delete_count`

The per-policy config section name must be of the form:

```
[proxy-server:policy:<policy index>]
```

Note: The per-policy config section name should refer to the policy index, not the policy name.

Note: The first part of proxy-server config section name must match the name of the proxy-server config section. This is typically `proxy-server` as shown above, but if different then the names of any per-policy config sections must be changed accordingly.

The value of an option specified in a per-policy section will override any value given in the proxy-server section for that policy only. Otherwise the value of these options will be that specified in the proxy-server section.

For example, the following section provides policy-specific options for a policy with index `3`:

```
[proxy-server:policy:3]
sorting_method = affinity
read_affinity = r2=1
write_affinity = r2
write_affinity_node_count = 1 * replicas
write_affinity_handoff_delete_count = 2
```

Note: It is recommended that per-policy config options are *not* included in the `[DEFAULT]` section. If they are then the following behavior applies.

Per-policy config sections will inherit options in the `[DEFAULT]` section of the config file, and any such inheritance will take precedence over inheriting options from the proxy-server config section.

Per-policy config section options will override options in the `[DEFAULT]` section. Unlike the behavior described under *General Server Configuration* for paste-deploy `filter` and `app` sections, the `set` keyword is not required for options to override in per-policy config sections.

For example, given the following settings in a config file:

```
[DEFAULT]
sorting_method = affinity
read_affinity = r0=100
write_affinity = r0

[app:proxy-server]
use = egg:swift#proxy
# use of set keyword here overrides [DEFAULT] option
set read_affinity = r1=100
# without set keyword, [DEFAULT] option overrides in a paste-deploy section
write_affinity = r1

[proxy-server:policy:0]
sorting_method = affinity
# set keyword not required here to override [DEFAULT] option
write_affinity = r1
```

would result in policy with index `0` having settings:

- `read_affinity = r0=100` (inherited from the `[DEFAULT]` section)
- `write_affinity = r1` (specified in the policy 0 section)

and any other policy would have the default settings of:

- `read_affinity = r1=100` (set in the proxy-server section)
- `write_affinity = r0` (inherited from the `[DEFAULT]` section)

Proxy Middlewares

Many features in Swift are implemented as middleware in the proxy-server pipeline. See *Middleware* and the `proxy-server.conf-sample` file for more information. In particular, the use of some type of *authentication and authorization middleware* is highly recommended.

5.2.8 Memcached Considerations

Several of the Services rely on Memcached for caching certain types of lookups, such as auth tokens, and container/account existence. Swift does not do any caching of actual object data. Memcached should be able to run on any servers that have available RAM and CPU. Typically Memcached is run on the proxy servers. The `memcache_servers` config option in the `proxy-server.conf` should contain all memcached servers.

Shard Range Listing Cache

When a container gets *sharded* the root container will still be the primary entry point to many container requests, as it provides the list of shards. To take load off the root container Swift by default caches the list of shards returned.

As the number of shards for a root container grows to more than 3k the memcache default max size of 1MB can be reached.

If you over-run your max configured memcache size youll see messages like:

```
Error setting value in memcached: 127.0.0.1:11211: SERVER_ERROR object too␣
↪large for cache
```

When you see these messages your root containers are getting hammered and probably returning 503 reponses to clients. Override the default 1MB limit to 5MB with something like:

```
/usr/bin/memcached -I 5000000 ...
```

Memcache has a `stats sizes` option that can point out the current size usage. As this reaches the current max an increase might be in order:

```
# telnet <memcache server> 11211
> stats sizes
STAT 160 2
STAT 448 1
STAT 576 1
END
```

5.2.9 System Time

Time may be relative but it is relatively important for Swift! Swift uses timestamps to determine which is the most recent version of an object. It is very important for the system time on each server in the cluster to by synced as closely as possible (more so for the proxy server, but in general it is a good idea for all the servers). Typical deployments use NTP with a local NTP server to ensure that the system times are as close as possible. This should also be monitored to ensure that the times do not vary too much.

5.2.10 General Service Tuning

Most services support either a `workers` or `concurrency` value in the settings. This allows the services to make effective use of the cores available. A good starting point is to set the concurrency level for the proxy and storage services to 2 times the number of cores available. If more than one service is sharing a server, then some experimentation may be needed to find the best balance.

For example, one operator reported using the following settings in a production Swift cluster:

- Proxy servers have dual quad core processors (i.e. 8 cores); testing has shown 16 workers to be a pretty good balance when saturating a 10g network and gives good CPU utilization.
- Storage server processes all run together on the same servers. These servers have dual quad core processors, for 8 cores total. The Account, Container, and Object servers are run with 8 workers each. Most of the background jobs are run at a concurrency of 1, with the exception of the replicators which are run at a concurrency of 2.

The `max_clients` parameter can be used to adjust the number of client requests an individual worker accepts for processing. The fewer requests being processed at one time, the less likely a request that consumes the workers CPU time, or blocks in the OS, will negatively impact other requests. The more requests being processed at one time, the more likely one worker can utilize network and disk capacity.

On systems that have more cores, and more memory, where one can afford to run more workers, raising the number of workers and lowering the maximum number of clients serviced per worker can lessen the impact of CPU intensive or stalled requests.

The `nice_priority` parameter can be used to set program scheduling priority. The `ionice_class` and `ionice_priority` parameters can be used to set I/O scheduling class and priority on the systems that use an I/O scheduler that supports I/O priorities. As at kernel 2.6.17 the only such scheduler is the Completely Fair Queuing (CFQ) I/O scheduler. If you run your Storage servers all together on the same servers, you can slow down the auditors or prioritize object-server I/O via these parameters (but probably do not need to change it on the proxy). It is a new feature and the best practices are still being developed. On some systems it may be required to run the daemons as root. For more info also see setpriority(2) and ioprio_set(2).

The above configuration setting should be taken as suggestions and testing of configuration settings should be done to ensure best utilization of CPU, network connectivity, and disk I/O.

5.2.11 Filesystem Considerations

Swift is designed to be mostly filesystem agnosticthe only requirement being that the filesystem supports extended attributes (xattrs). After thorough testing with our use cases and hardware configurations, XFS was the best all-around choice. If you decide to use a filesystem other than XFS, we highly recommend thorough testing.

For distros with more recent kernels (for example Ubuntu 12.04 Precise), we recommend using the default settings (including the default inode size of 256 bytes) when creating the file system:

```
mkfs.xfs -L D1 /dev/sda1
```

In the last couple of years, XFS has made great improvements in how inodes are allocated and used. Using the default inode size no longer has an impact on performance.

For distros with older kernels (for example Ubuntu 10.04 Lucid), some settings can dramatically impact performance. We recommend the following when creating the file system:

```
mkfs.xfs -i size=1024 -L D1 /dev/sda1
```

Setting the inode size is important, as XFS stores xattr data in the inode. If the metadata is too large to fit in the inode, a new extent is created, which can cause quite a performance problem. Upping the inode size to 1024 bytes provides enough room to write the default metadata, plus a little headroom.

The following example mount options are recommended when using XFS:

```
mount -t xfs -o noatime -L D1 /srv/node/d1
```

We do not recommend running Swift on RAID, but if you are using RAID it is also important to make sure that the proper sunit and swidth settings get set so that XFS can make most efficient use of the RAID array.

For a standard Swift install, all data drives are mounted directly under `/srv/node` (as can be seen in the above example of mounting label `D1` as `/srv/node/d1`). If you choose to mount the drives in another directory, be sure to set the `devices` config option in all of the server configs to point to the correct directory.

The mount points for each drive in `/srv/node/` should be owned by the root user almost exclusively (`root:root 755`). This is required to prevent rsync from syncing files into the root drive in the event a drive is unmounted.

Swift uses system calls to reserve space for new objects being written into the system. If your filesystem does not support `fallocate()` or `posix_fallocate()`, be sure to set the `disable_fallocate = true` config parameter in account, container, and object server configs.

Most current Linux distributions ship with a default installation of updatedb. This tool runs periodically and updates the file name database that is used by the GNU locate tool. However, including Swift object and container database files is most likely not required and the periodic update affects the performance quite a bit. To disable the inclusion of these files add the path where Swift stores its data to the setting PRUNEPATHS in `/etc/updatedb.conf`:

```
PRUNEPATHS="... /tmp ... /var/spool ... /srv/node"
```

5.2.12 General System Tuning

The following changes have been found to be useful when running Swift on Ubuntu Server 10.04.

The following settings should be in `/etc/sysctl.conf`:

```
# disable TIME_WAIT.. wait..
net.ipv4.tcp_tw_recycle=1
net.ipv4.tcp_tw_reuse=1

# disable syn cookies
net.ipv4.tcp_syncookies = 0

# double amount of allowed conntrack
net.netfilter.nf_conntrack_max = 262144
```

To load the updated sysctl settings, run `sudo sysctl -p`.

A note about changing the TIME_WAIT values. By default the OS will hold a port open for 60 seconds to ensure that any remaining packets can be received. During high usage, and with the number of connections that are created, it is easy to run out of ports. We can change this since we are in control of the network. If you are not in control of the network, or do not expect high loads, then you may not want to adjust those values.

5.2.13 Logging Considerations

Swift is set up to log directly to syslog. Every service can be configured with the `log_facility` option to set the syslog log facility destination. We recommended using syslog-ng to route the logs to specific log files locally on the server and also to remote log collecting servers. Additionally, custom log handlers can be used via the custom_log_handlers setting.

5.3 Apache Deployment Guide

5.3.1 Web Front End Considerations

Swift can be configured to work both using an integral web front-end and using a full-fledged Web Server such as the Apache2 (HTTPD) web server. The integral web front-end is a wsgi mini Web Server which opens up its own socket and serves http requests directly. The incoming requests accepted by the integral web front-end are then forwarded to a wsgi application (the core swift) for further handling, possibly via wsgi middleware sub-components.

client<->integral web front-end<->middleware<->core swift

To gain full advantage of Apache2, Swift can alternatively be configured to work as a request processor of the Apache2 server. This alternative deployment scenario uses mod_wsgi of Apache2 to forward requests to the swift wsgi application and middleware.

client<->Apache2 with mod_wsgi<>middleware<->core swift

The integral web front-end offers simplicity and requires minimal configuration. It is also the web front-end most commonly used with Swift. Additionally, the integral web front-end includes support for re-

ceiving chunked transfer encoding from a client, presently not supported by Apache2 in the operation mode described here.

The use of Apache2 offers new ways to extend Swift and integrate it with existing authentication, administration and control systems. A single Apache2 server can serve as the web front end of any number of swift servers residing on a swift node. For example when a storage node offers account, container and object services, a single Apache2 server can serve as the web front end of all three services.

The apache variant described here was tested as part of an IBM research work. It was found that following tuning, the Apache2 offer generally equivalent performance to that offered by the integral web front-end. Alternative to Apache2, other web servers may be used, but were never tested.

5.3.2 Apache2 Setup

Both Apache2 and mod-wsgi needs to be installed on the system. Ubuntu comes with Apache2 installed. Install mod-wsgi using:

```
sudo apt-get install libapache2-mod-wsgi
```

Create a directory for the Apache2 wsgi files:

```
sudo mkdir /srv/www/swift
```

Create a working directory for the wsgi processes:

```
sudo mkdir -m 2770 /var/lib/swift
sudo chown swift:swift /var/lib/swift
```

Create a file for each service under `/srv/www/swift`.

For a proxy service create `/srv/www/swift/proxy-server.wsgi`:

```
from swift.common.wsgi import init_request_processor
application, conf, logger, log_name = \
    init_request_processor('/etc/swift/proxy-server.conf','proxy-server')
```

For an account service create `/srv/www/swift/account-server.wsgi`:

```
from swift.common.wsgi import init_request_processor
application, conf, logger, log_name = \
    init_request_processor('/etc/swift/account-server.conf',
                           'account-server')
```

For an container service create `/srv/www/swift/container-server.wsgi`:

```
from swift.common.wsgi import init_request_processor
application, conf, logger, log_name = \
    init_request_processor('/etc/swift/container-server.conf',
                          'container-server')
```

For an object service create `/srv/www/swift/object-server.wsgi`:

```
from swift.common.wsgi import init_request_processor
application, conf, logger, log_name = \
    init_request_processor('/etc/swift/object-server.conf',
                           'object-server')
```

Create a `/etc/apache2/conf.d/swift_wsgi.conf` configuration file that will define a port and Virtual Host per each local service. For example an Apache2 serving as a web front end of a proxy service:

```
# Proxy
Listen 8080

<VirtualHost *:8080>
    ServerName proxy-server

    LimitRequestBody 5368709122
    LimitRequestFields 200

    WSGIDaemonProcess proxy-server processes=5 threads=1 user=swift␣
↪group=swift display-name=%{GROUP}
    WSGIProcessGroup proxy-server
    WSGIScriptAlias / /srv/www/swift/proxy-server.wsgi
    LogLevel debug
    CustomLog /var/log/apache2/proxy.log combined
    ErrorLog /var/log/apache2/proxy-server
</VirtualHost>
```

Notice that when using Apache the limit on the maximal object size should be imposed by Apache using the *LimitRequestBody* rather by the swift proxy. Note also that the *LimitRequestBody* should indicate the same value as indicated by *max_file_size* located in both `/etc/swift/swift.conf` and in `/etc/swift/test.conf`. The Swift default value for *max_file_size* (when not present) is *5368709122*. For example an Apache2 serving as a web front end of a storage node:

```
# Object Service
Listen 6200

<VirtualHost *:6200>
    ServerName object-server

    LimitRequestFields 200

    WSGIDaemonProcess object-server processes=5 threads=1 user=swift␣
↪group=swift display-name=%{GROUP}
    WSGIProcessGroup object-server
    WSGIScriptAlias / /srv/www/swift/object-server.wsgi
    LogLevel debug
    CustomLog /var/log/apache2/access.log combined
    ErrorLog /var/log/apache2/object-server
</VirtualHost>

# Container Service
```

(continues on next page)

(continued from previous page)

```
Listen 6201

<VirtualHost *:6201>
    ServerName container-server

    LimitRequestFields 200

    WSGIDaemonProcess container-server processes=5 threads=1 user=swift␣
↪group=swift display-name=%{GROUP}
    WSGIProcessGroup container-server
    WSGIScriptAlias / /srv/www/swift/container-server.wsgi
    LogLevel debug
    CustomLog /var/log/apache2/access.log combined
    ErrorLog /var/log/apache2/container-server
</VirtualHost>

# Account Service
Listen 6202

<VirtualHost *:6202>
    ServerName account-server

    LimitRequestFields 200

    WSGIDaemonProcess account-server processes=5 threads=1 user=swift␣
↪group=swift display-name=%{GROUP}
    WSGIProcessGroup account-server
    WSGIScriptAlias / /srv/www/swift/account-server.wsgi
    LogLevel debug
    CustomLog /var/log/apache2/access.log combined
    ErrorLog /var/log/apache2/account-server
</VirtualHost>
```

Enable the newly configured Virtual Hosts:

```
a2ensite swift_wsgi.conf
```

Next, stop, test and start Apache2 again:

```
# stop it
systemctl stop apache2.service

# test the configuration
apache2ctl -t

# start it if the test succeeds
systemctl start apache2.service
```

Edit the tests config file and add:

```
web_front_end = apache2
normalized_urls = True
```

Also check to see that the file includes *max_file_size* of the same value as used for the *LimitRequestBody* in the apache config file above.

We are done. You may run functional tests to test - e.g.:

```
cd ~swift/swift
./.functests
```

5.4 Administrators Guide

5.4.1 Defining Storage Policies

Defining your Storage Policies is very easy to do with Swift. It is important that the administrator understand the concepts behind Storage Policies before actually creating and using them in order to get the most benefit out of the feature and, more importantly, to avoid having to make unnecessary changes once a set of policies have been deployed to a cluster.

It is highly recommended that the reader fully read and comprehend *Storage Policies* before proceeding with administration of policies. Plan carefully and it is suggested that experimentation be done first on a non-production cluster to be certain that the desired configuration meets the needs of the users. See *Upgrading and Confirming Functionality* before planning the upgrade of your existing deployment.

Following is a high level view of the very few steps it takes to configure policies once you have decided what you want to do:

1. Define your policies in `/etc/swift/swift.conf`
2. Create the corresponding object rings
3. Communicate the names of the Storage Policies to cluster users

For a specific example that takes you through these steps, please see *Adding Storage Policies to an Existing SAIO*

5.4.2 Managing the Rings

You may build the storage rings on any server with the appropriate version of Swift installed. Once built or changed (rebalanced), you must distribute the rings to all the servers in the cluster. Storage rings contain information about all the Swift storage partitions and how they are distributed between the different nodes and disks.

Swift 1.6.0 is the last version to use a Python pickle format. Subsequent versions use a different serialization format. **Rings generated by Swift versions 1.6.0 and earlier may be read by any version, but rings generated after 1.6.0 may only be read by Swift versions greater than 1.6.0.** So when upgrading from version 1.6.0 or earlier to a version greater than 1.6.0, either upgrade Swift on your ring building server **last** after all Swift nodes have been successfully upgraded, or refrain from generating rings until all Swift nodes have been successfully upgraded.

If you need to downgrade from a version of Swift greater than 1.6.0 to a version less than or equal to 1.6.0, first downgrade your ring-building server, generate new rings, push them out, then continue with the rest of the downgrade.

For more information see *The Rings*.

Removing a device from the ring:

```
swift-ring-builder <builder-file> remove <ip_address>/<device_name>
```

Removing a server from the ring:

```
swift-ring-builder <builder-file> remove <ip_address>
```

Adding devices to the ring:

See *Preparing the Ring*

See what devices for a server are in the ring:

```
swift-ring-builder <builder-file> search <ip_address>
```

Once you are done with all changes to the ring, the changes need to be committed:

```
swift-ring-builder <builder-file> rebalance
```

Once the new rings are built, they should be pushed out to all the servers in the cluster.

Optionally, if invoked as swift-ring-builder-safe the directory containing the specified builder file will be locked (via a .lock file in the parent directory). This provides a basic safe guard against multiple instances of the swift-ring-builder (or other utilities that observe this lock) from attempting to write to or read the builder/ring files while operations are in progress. This can be useful in environments where ring management has been automated but the operator still needs to interact with the rings manually.

If the ring builder is not producing the balances that you are expecting, you can gain visibility into what its doing with the `--debug` flag.:

```
swift-ring-builder <builder-file> rebalance --debug
```

This produces a great deal of output that is mostly useful if you are either (a) attempting to fix the ring builder, or (b) filing a bug against the ring builder.

You may notice in the rebalance output a dispersion number. What this number means is explained in *Dispersion* but in essence is the percentage of partitions in the ring that have too many replicas within a particular failure domain. You can ask swift-ring-builder what the dispersion is with:

```
swift-ring-builder <builder-file> dispersion
```

This will give you the percentage again, if you want a detailed view of the dispersion simply add a `--verbose`:

```
swift-ring-builder <builder-file> dispersion --verbose
```

This will not only display the percentage but will also display a dispersion table that lists partition dispersion by tier. You can use this table to figure out were you need to add capacity or to help tune an *Overload* value.

Now lets take an example with 1 region, 3 zones and 4 devices. Each device has the same weight, and the `dispersion --verbose` might show the following:

```
Dispersion is 16.666667, Balance is 0.000000, Overload is 0.00%
Required overload is 33.333333%
Worst tier is 33.333333 (r1z3)
-----------------------------------------------------------------------
Tier                         Parts      %    Max     0     1     2     3
-----------------------------------------------------------------------
r1                             768   0.00      3     0     0     0   256
r1z1                           192   0.00      1    64   192     0     0
r1z1-127.0.0.1                 192   0.00      1    64   192     0     0
r1z1-127.0.0.1/sda             192   0.00      1    64   192     0     0
r1z2                           192   0.00      1    64   192     0     0
r1z2-127.0.0.2                 192   0.00      1    64   192     0     0
r1z2-127.0.0.2/sda             192   0.00      1    64   192     0     0
r1z3                           384  33.33      1     0   128   128     0
r1z3-127.0.0.3                 384  33.33      1     0   128   128     0
r1z3-127.0.0.3/sda             192   0.00      1    64   192     0     0
r1z3-127.0.0.3/sdb             192   0.00      1    64   192     0     0
```

The first line reports that there are 256 partitions with 3 copies in region 1; and this is an expected output in this case (single region with 3 replicas) as reported by the Max value.

However, there is some imbalance in the cluster, more precisely in zone 3. The Max reports a maximum of 1 copy in this zone; however 50.00% of the partitions are storing 2 replicas in this zone (which is somewhat expected, because there are more disks in this zone).

You can now either add more capacity to the other zones, decrease the total weight in zone 3 or set the overload to a value *greater than* 33.333333% - only as much overload as needed will be used.

5.4.3 Scripting Ring Creation

You can create scripts to create the account and container rings and rebalance. Heres an example script for the Account ring. Use similar commands to create a make-container-ring.sh script on the proxy server node.

1. Create a script file called make-account-ring.sh on the proxy server node with the following content:

   ```
   #!/bin/bash
   cd /etc/swift
   rm -f account.builder account.ring.gz backups/account.builder backups/
   ↪account.ring.gz
   swift-ring-builder account.builder create 18 3 1
   swift-ring-builder account.builder add r1z1-<account-server-1>:6202/sdb1 1
   swift-ring-builder account.builder add r1z2-<account-server-2>:6202/sdb1 1
   swift-ring-builder account.builder rebalance
   ```

   You need to replace the values of <account-server-1>, <account-server-2>, etc. with the IP addresses of the account servers used in your setup. You can have as many account servers as you need. All account servers are assumed to be listening on port 6202, and have a storage device called sdb1 (this is a directory name created under /drives when we setup the account server). The

z1, z2, etc. designate zones, and you can choose whether you put devices in the same or different zones. The r1 designates the region, with different regions specified as r1, r2, etc.

2. Make the script file executable and run it to create the account ring file:

```
chmod +x make-account-ring.sh
sudo ./make-account-ring.sh
```

3. Copy the resulting ring file /etc/swift/account.ring.gz to all the account server nodes in your Swift environment, and put them in the /etc/swift directory on these nodes. Make sure that every time you change the account ring configuration, you copy the resulting ring file to all the account nodes.

5.4.4 Handling System Updates

It is recommended that system updates and reboots are done a zone at a time. This allows the update to happen, and for the Swift cluster to stay available and responsive to requests. It is also advisable when updating a zone, let it run for a while before updating the other zones to make sure the update doesnt have any adverse effects.

5.4.5 Handling Drive Failure

In the event that a drive has failed, the first step is to make sure the drive is unmounted. This will make it easier for Swift to work around the failure until it has been resolved. If the drive is going to be replaced immediately, then it is just best to replace the drive, format it, remount it, and let replication fill it up.

After the drive is unmounted, make sure the mount point is owned by root (root:root 755). This ensures that rsync will not try to replicate into the root drive once the failed drive is unmounted.

If the drive cant be replaced immediately, then it is best to leave it unmounted, and set the device weight to 0. This will allow all the replicas that were on that drive to be replicated elsewhere until the drive is replaced. Once the drive is replaced, the device weight can be increased again. Setting the device weight to 0 instead of removing the drive from the ring gives Swift the chance to replicate data from the failing disk too (in case it is still possible to read some of the data).

Setting the device weight to 0 (or removing a failed drive from the ring) has another benefit: all partitions that were stored on the failed drive are distributed over the remaining disks in the cluster, and each disk only needs to store a few new partitions. This is much faster compared to replicating all partitions to a single, new disk. It decreases the time to recover from a degraded number of replicas significantly, and becomes more and more important with bigger disks.

5.4.6 Handling Server Failure

If a server is having hardware issues, it is a good idea to make sure the Swift services are not running. This will allow Swift to work around the failure while you troubleshoot.

If the server just needs a reboot, or a small amount of work that should only last a couple of hours, then it is probably best to let Swift work around the failure and get the machine fixed and back online. When the machine comes back online, replication will make sure that anything that is missing during the downtime will get updated.

If the server has more serious issues, then it is probably best to remove all of the servers devices from the ring. Once the server has been repaired and is back online, the servers devices can be added back into

the ring. It is important that the devices are reformatted before putting them back into the ring as it is likely to be responsible for a different set of partitions than before.

5.4.7 Detecting Failed Drives

It has been our experience that when a drive is about to fail, error messages will spew into */var/log/kern.log*. There is a script called *swift-drive-audit* that can be run via cron to watch for bad drives. If errors are detected, it will unmount the bad drive, so that Swift can work around it. The script takes a configuration file with the following settings:

`[drive-audit]`

Option	Default	Description
user	swift	Drop privileges to this user for non-root tasks
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Log level
device_dir	/srv/node	Directory devices are mounted under
minutes	60	Number of minutes to look back in */var/log/kern.log*
error_limit	1	Number of errors to find before a device is unmounted
log_file_pattern	/var/log/kern*	Location of the log file with globbing pattern to check against device errors
regex_pattern_X	(see below)	Regular expression patterns to be used to locate device blocks with errors in the log file

The default regex pattern used to locate device blocks with errors are *berrorb.*b(sd[a-z]{1,2}d?)b* and *b(sd[a-z]{1,2}d?)b.*berrorb*. One is able to overwrite the default above by providing new expressions using the format *regex_pattern_X = regex_expression*, where *X* is a number.

This script has been tested on Ubuntu 10.04 and Ubuntu 12.04, so if you are using a different distro or OS, some care should be taken before using in production.

5.4.8 Preventing Disk Full Scenarios

Prevent disk full scenarios by ensuring that the `proxy-server` blocks PUT requests and rsync prevents replication to the specific drives.

You can prevent *proxy-server* PUT requests to low space disks by ensuring `fallocate_reserve` is set in `account-server.conf`, `container-server.conf`, and `object-server.conf`. By default, `fallocate_reserve` is set to 1%. In the object server, this blocks PUT requests that would leave the free disk space below 1% of the disk. In the account and container servers, this blocks operations that will increase account or container database size once the free disk space falls below 1%.

Setting `fallocate_reserve` is highly recommended to avoid filling disks to 100%. When Swifts disks are completely full, all requests involving those disks will fail, including DELETE requests that would otherwise free up space. This is because object deletion includes the creation of a zero-byte tombstone (.ts) to record the time of the deletion for replication purposes; this happens prior to deletion of the objects data. On a completely-full filesystem, that zero-byte .ts file cannot be created, so the DELETE request will fail and the disk will remain completely full. If `fallocate_reserve` is set, then the filesystem will have enough space to create the zero-byte .ts file, and thus the deletion of the object will succeed and free up some space.

In order to prevent rsync replication to specific drives, firstly setup `rsync_module` per disk in your `object-replicator`. Set this in `object-server.conf`:

```
[object-replicator]
rsync_module = {replication_ip}::object_{device}
```

Set the individual drives in `rsync.conf`. For example:

```
[object_sda]
max connections = 4
lock file = /var/lock/object_sda.lock

[object_sdb]
max connections = 4
lock file = /var/lock/object_sdb.lock
```

Finally, monitor the disk space of each disk and adjust the rsync `max connections` per drive to `-1`. We recommend utilising your existing monitoring solution to achieve this. The following is an example script:

```
#!/usr/bin/env python
import os
import errno

RESERVE = 500 * 2 ** 20  # 500 MiB

DEVICES = '/srv/node1'

path_template = '/etc/rsync.d/disable_%s.conf'
config_template = '''
[object_%s]
max connections = -1
'''

def disable_rsync(device):
    with open(path_template % device, 'w') as f:
        f.write(config_template.lstrip() % device)


def enable_rsync(device):
    try:
        os.unlink(path_template % device)
    except OSError as e:
        # ignore file does not exist
        if e.errno != errno.ENOENT:
            raise


for device in os.listdir(DEVICES):
    path = os.path.join(DEVICES, device)
    st = os.statvfs(path)
```

(continues on next page)

(continued from previous page)

```
    free = st.f_bavail * st.f_frsize
    if free < RESERVE:
        disable_rsync(device)
    else:
        enable_rsync(device)
```

For the above script to work, ensure `/etc/rsync.d/` conf files are included, by specifying `&include` in your `rsync.conf` file:

```
&include /etc/rsync.d
```

Use this in conjunction with a cron job to periodically run the script, for example:

```
# /etc/cron.d/devicecheck
* * * * * root /some/path/to/disable_rsync.py
```

5.4.9 Dispersion Report

There is a swift-dispersion-report tool for measuring overall cluster health. This is accomplished by checking if a set of deliberately distributed containers and objects are currently in their proper places within the cluster.

For instance, a common deployment has three replicas of each object. The health of that object can be measured by checking if each replica is in its proper place. If only 2 of the 3 is in place the objects heath can be said to be at 66.66%, where 100% would be perfect.

A single objects health, especially an older object, usually reflects the health of that entire partition the object is in. If we make enough objects on a distinct percentage of the partitions in the cluster, we can get a pretty valid estimate of the overall cluster health. In practice, about 1% partition coverage seems to balance well between accuracy and the amount of time it takes to gather results.

The first thing that needs to be done to provide this health value is create a new account solely for this usage. Next, we need to place the containers and objects throughout the system so that they are on distinct partitions. The swift-dispersion-populate tool does this by making up random container and object names until they fall on distinct partitions. Last, and repeatedly for the life of the cluster, we need to run the swift-dispersion-report tool to check the health of each of these containers and objects.

These tools need direct access to the entire cluster and to the ring files (installing them on a proxy server will probably do). Both swift-dispersion-populate and swift-dispersion-report use the same configuration file, /etc/swift/dispersion.conf. Example conf file:

```
[dispersion]
auth_url = http://localhost:8080/auth/v1.0
auth_user = test:tester
auth_key = testing
endpoint_type = internalURL
```

There are also options for the conf file for specifying the dispersion coverage (defaults to 1%), retries, concurrency, etc. though usually the defaults are fine. If you want to use keystone v3 for authentication there are options like auth_version, user_domain_name, project_domain_name and project_name.

Once the configuration is in place, run *swift-dispersion-populate* to populate the containers and objects throughout the cluster.

Now that those containers and objects are in place, you can run *swift-dispersion-report* to get a dispersion report, or the overall health of the cluster. Here is an example of a cluster in perfect health:

```
$ swift-dispersion-report
Queried 2621 containers for dispersion reporting, 19s, 0 retries
100.00% of container copies found (7863 of 7863)
Sample represents 1.00% of the container partition space

Queried 2619 objects for dispersion reporting, 7s, 0 retries
100.00% of object copies found (7857 of 7857)
Sample represents 1.00% of the object partition space
```

Now Ill deliberately double the weight of a device in the object ring (with replication turned off) and rerun the dispersion report to show what impact that has:

```
$ swift-ring-builder object.builder set_weight d0 200
$ swift-ring-builder object.builder rebalance
...
$ swift-dispersion-report
Queried 2621 containers for dispersion reporting, 8s, 0 retries
100.00% of container copies found (7863 of 7863)
Sample represents 1.00% of the container partition space

Queried 2619 objects for dispersion reporting, 7s, 0 retries
There were 1763 partitions missing one copy.
77.56% of object copies found (6094 of 7857)
Sample represents 1.00% of the object partition space
```

You can see the health of the objects in the cluster has gone down significantly. Of course, I only have four devices in this test environment, in a production environment with many many devices the impact of one device change is much less. Next, Ill run the replicators to get everything put back into place and then rerun the dispersion report:

```
... start object replicators and monitor logs until they're caught up ...
$ swift-dispersion-report
Queried 2621 containers for dispersion reporting, 17s, 0 retries
100.00% of container copies found (7863 of 7863)
Sample represents 1.00% of the container partition space

Queried 2619 objects for dispersion reporting, 7s, 0 retries
100.00% of object copies found (7857 of 7857)
Sample represents 1.00% of the object partition space
```

You can also run the report for only containers or objects:

```
$ swift-dispersion-report --container-only
Queried 2621 containers for dispersion reporting, 17s, 0 retries
100.00% of container copies found (7863 of 7863)
Sample represents 1.00% of the container partition space
```

(continues on next page)

(continued from previous page)

```
$ swift-dispersion-report --object-only
Queried 2619 objects for dispersion reporting, 7s, 0 retries
100.00% of object copies found (7857 of 7857)
Sample represents 1.00% of the object partition space
```

Alternatively, the dispersion report can also be output in JSON format. This allows it to be more easily consumed by third party utilities:

```
$ swift-dispersion-report -j
{"object": {"retries:": 0, "missing_two": 0, "copies_found": 7863, "missing_
↪one": 0, "copies_expected": 7863, "pct_found": 100.0, "overlapping": 0,
↪"missing_all": 0}, "container": {"retries:": 0, "missing_two": 0, "copies_
↪found": 12534, "missing_one": 0, "copies_expected": 12534, "pct_found": 100.
↪0, "overlapping": 15, "missing_all": 0}}
```

Note that you may select which storage policy to use by setting the option policy-name silver or -P silver (silver is the example policy name here). If no policy is specified, the default will be used per the swift.conf file. When you specify a policy the containers created also include the policy index, thus even when running a container_only report, you will need to specify the policy not using the default.

5.4.10 Geographically Distributed Swift Considerations

Swift provides two features that may be used to distribute replicas of objects across multiple geographically distributed data-centers: with *Global Clusters* object replicas may be dispersed across devices from different data-centers by using *regions* in ring device descriptors; with *Container to Container Synchronization* objects may be copied between independent Swift clusters in each data-center. The operation and configuration of each are described in their respective documentation. The following points should be considered when selecting the feature that is most appropriate for a particular use case:

1. Global Clusters allows the distribution of object replicas across data-centers to be controlled by the cluster operator on per-policy basis, since the distribution is determined by the assignment of devices from each data-center in each policys ring file. With Container Sync the end user controls the distribution of objects across clusters on a per-container basis.

2. Global Clusters requires an operator to coordinate ring deployments across multiple data-centers. Container Sync allows for independent management of separate Swift clusters in each data-center, and for existing Swift clusters to be used as peers in Container Sync relationships without deploying new policies/rings.

3. Global Clusters seamlessly supports features that may rely on cross-container operations such as large objects and versioned writes. Container Sync requires the end user to ensure that all required containers are syncd for these features to work in all data-centers.

4. Global Clusters makes objects available for GET or HEAD requests in both data-centers even if a replica of the object has not yet been asynchronously migrated between data-centers, by forwarding requests between data-centers. Container Sync is unable to serve requests for an object in a particular data-center until the asynchronous sync process has copied the object to that data-center.

5. Global Clusters may require less storage capacity than Container Sync to achieve equivalent durability of objects in each data-center. Global Clusters can restore replicas that are lost or corrupted in one data-center using replicas from other data-centers. Container Sync requires each data-center

to independently manage the durability of objects, which may result in each data-center storing more replicas than with Global Clusters.

6. Global Clusters execute all account/container metadata updates synchronously to account/container replicas in all data-centers, which may incur delays when making updates across WANs. Container Sync only copies objects between data-centers and all Swift internal traffic is confined to each data-center.

7. Global Clusters does not yet guarantee the availability of objects stored in Erasure Coded policies when one data-center is offline. With Container Sync the availability of objects in each data-center is independent of the state of other data-centers once objects have been synced. Container Sync also allows objects to be stored using different policy types in different data-centers.

Checking handoff partition distribution

You can check if handoff partitions are piling up on a server by comparing the expected number of partitions with the actual number on your disks. First get the number of partitions that are currently assigned to a server using the `dispersion` command from `swift-ring-builder`:

```
swift-ring-builder sample.builder dispersion --verbose
Dispersion is 0.000000, Balance is 0.000000, Overload is 0.00%
Required overload is 0.000000%
------------------------------------------------------------------------------
Tier                            Parts      %    Max     0     1     2     3
------------------------------------------------------------------------------
r1                               8192   0.00      2     0     0  8192     0
r1z1                             4096   0.00      1  4096  4096     0     0
r1z1-172.16.10.1                 4096   0.00      1  4096  4096     0     0
r1z1-172.16.10.1/sda1            4096   0.00      1  4096  4096     0     0
r1z2                             4096   0.00      1  4096  4096     0     0
r1z2-172.16.10.2                 4096   0.00      1  4096  4096     0     0
r1z2-172.16.10.2/sda1            4096   0.00      1  4096  4096     0     0
r1z3                             4096   0.00      1  4096  4096     0     0
r1z3-172.16.10.3                 4096   0.00      1  4096  4096     0     0
r1z3-172.16.10.3/sda1            4096   0.00      1  4096  4096     0     0
r1z4                             4096   0.00      1  4096  4096     0     0
r1z4-172.16.20.4                 4096   0.00      1  4096  4096     0     0
r1z4-172.16.20.4/sda1            4096   0.00      1  4096  4096     0     0
r2                               8192   0.00      2     0  8192     0     0
r2z1                             4096   0.00      1  4096  4096     0     0
r2z1-172.16.20.1                 4096   0.00      1  4096  4096     0     0
r2z1-172.16.20.1/sda1            4096   0.00      1  4096  4096     0     0
r2z2                             4096   0.00      1  4096  4096     0     0
r2z2-172.16.20.2                 4096   0.00      1  4096  4096     0     0
r2z2-172.16.20.2/sda1            4096   0.00      1  4096  4096     0     0
```

As you can see from the output, each server should store 4096 partitions, and each region should store 8192 partitions. This example used a partition power of 13 and 3 replicas.

With write_affinity enabled it is expected to have a higher number of partitions on disk compared to the value reported by the swift-ring-builder dispersion command. The number of additional (handoff) partitions in region r1 depends on your cluster size, the amount of incoming data as well as the replication

speed.

Lets use the example from above with 6 nodes in 2 regions, and write_affinity configured to write to region r1 first. *swift-ring-builder* reported that each node should store 4096 partitions:

```
Expected partitions for region r2:                                    8192
Handoffs stored across 4 nodes in region r1:                  8192 / 4 =ă2048
Maximum number of partitions on each server in region r1: 2048 + 4096 = 6144
```

Worst case is that handoff partitions in region 1 are populated with new object replicas faster than replication is able to move them to region 2. In that case you will see ~ 6144 partitions per server in region r1. Your actual number should be lower and between 4096 and 6144 partitions (preferably on the lower side).

Now count the number of object partitions on a given server in region 1, for example on 172.16.10.1. Note that the pathnames might be different; */srv/node/* is the default mount location, and *objects* applies only to storage policy 0 (storage policy 1 would use *objects-1* and so on):

```
find -L /srv/node/ -maxdepth 3 -type d -wholename "*objects/*" | wc -l
```

If this number is always on the upper end of the expected partition number range (4096 to 6144) or increasing you should check your replication speed and maybe even disable write_affinity. Please refer to the next section how to collect metrics from Swift, and especially *swift-recon -r* how to check replication stats.

5.4.11 Cluster Telemetry and Monitoring

Various metrics and telemetry can be obtained from the account, container, and object servers using the recon server middleware and the swift-recon cli. To do so update your account, container, or object servers pipelines to include recon and add the associated filter config.

object-server.conf sample:

```
[pipeline:main]
pipeline = recon object-server

[filter:recon]
use = egg:swift#recon
recon_cache_path = /var/cache/swift
```

container-server.conf sample:

```
[pipeline:main]
pipeline = recon container-server

[filter:recon]
use = egg:swift#recon
recon_cache_path = /var/cache/swift
```

account-server.conf sample:

```
[pipeline:main]
pipeline = recon account-server

[filter:recon]
use = egg:swift#recon
recon_cache_path = /var/cache/swift
```

The recon_cache_path simply sets the directory where stats for a few items will be stored. Depending on the method of deployment you may need to create this directory manually and ensure that Swift has read/write access.

Finally, if you also wish to track asynchronous pending on your object servers you will need to setup a cronjob to run the swift-recon-cron script periodically on your object servers:

```
*/5 * * * * swift /usr/bin/swift-recon-cron /etc/swift/object-server.conf
```

Once the recon middleware is enabled, a GET request for /recon/<metric> to the backend object server will return a JSON-formatted response:

```
fhines@ubuntu:~$ curl -i http://localhost:6230/recon/async
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 20
Date: Tue, 18 Oct 2011 21:03:01 GMT

{"async_pending": 0}
```

Note that the default port for the object server is 6200, except on a Swift All-In-One installation, which uses 6210, 6220, 6230, and 6240.

The following metrics and telemetry are currently exposed:

Request URI	Description
/recon/load	returns 1,5, and 15 minute load average
/recon/mem	returns /proc/meminfo
/recon/mounted	returns *ALL* currently mounted filesystems
/recon/unmounted	returns all unmounted drives if mount_check = True
/recon/diskusage	returns disk utilization for storage devices
/recon/driveaudit	returns # of drive audit errors
/recon/ringmd5	returns object/container/account ring md5sums
/recon/swiftconfmd5	returns swift.conf md5sum
/recon/quarantined	returns # of quarantined objects/accounts/containers
/recon/sockstat	returns consumable info from /proc/net/sockstat|6
/recon/devices	returns list of devices and devices dir i.e. /srv/node
/recon/async	returns count of async pending
/recon/replication	returns object replication info (for backward compatibility)
/recon/replication/<type>	returns replication info for given type (account, container, object)
/recon/auditor/<type>	returns auditor stats on last reported scan for given type (account, container, object)
/recon/updater/<type>	returns last updater sweep times for given type (container, object)
/recon/expirer/object	returns time elapsed and number of objects deleted during last object expirer sweep
/recon/version	returns Swift version
/recon/time	returns node time

Note that object_replication_last and object_replication_time in object replication info are considered to be transitional and will be removed in the subsequent releases. Use replication_last and replication_time instead.

This information can also be queried via the swift-recon command line utility:

```
fhines@ubuntu:~$ swift-recon -h
Usage:
        usage: swift-recon <server_type> [-v] [--suppress] [-a] [-r] [-u] [-d]
        [-R] [-l] [-T] [--md5] [--auditor] [--updater] [--expirer] [--
↪sockstat]

        <server_type>   account|container|object
        Defaults to object server.

        ex: swift-recon container -l --auditor


Options:
  -h, --help            show this help message and exit
  -v, --verbose         Print verbose info
  --suppress            Suppress most connection related errors
  -a, --async           Get async stats
  -r, --replication     Get replication stats
  -R, --reconstruction  Get reconstruction stats
  --auditor             Get auditor stats
```

(continues on next page)

(continued from previous page)

```
  --updater             Get updater stats
  --expirer             Get expirer stats
  -u, --unmounted       Check cluster for unmounted devices
  -d, --diskusage       Get disk usage stats
  -l, --loadstats       Get cluster load average stats
  -q, --quarantined     Get cluster quarantine stats
  --md5                 Get md5sum of servers ring and compare to local copy
  --sockstat            Get cluster socket usage stats
  -T, --time            Check time synchronization
  --all                 Perform all checks. Equal to
                        -arudlqT --md5 --sockstat --auditor --updater
                        --expirer --driveaudit --validate-servers
  -z ZONE, --zone=ZONE  Only query servers in specified zone
  -t SECONDS, --timeout=SECONDS
                        Time to wait for a response from a server
  --swiftdir=SWIFTDIR   Default = /etc/swift
```

For example, to obtain container replication info from all hosts in zone 3:

```
fhines@ubuntu:~$ swift-recon container -r --zone 3
===============================================================================
--> Starting reconnaissance on 1 hosts
===============================================================================
[2012-04-02 02:45:48] Checking on replication
[failure] low: 0.000, high: 0.000, avg: 0.000, reported: 1
[success] low: 486.000, high: 486.000, avg: 486.000, reported: 1
[replication_time] low: 20.853, high: 20.853, avg: 20.853, reported: 1
[attempted] low: 243.000, high: 243.000, avg: 243.000, reported: 1
```

5.4.12 Reporting Metrics to StatsD

If you have a StatsD server running, Swift may be configured to send it real-time operational metrics. To enable this, set the following configuration entries (see the sample configuration files):

```
log_statsd_host = localhost
log_statsd_port = 8125
log_statsd_default_sample_rate = 1.0
log_statsd_sample_rate_factor = 1.0
log_statsd_metric_prefix =                [empty-string]
```

If *log_statsd_host* is not set, this feature is disabled. The default values for the other settings are given above. The *log_statsd_host* can be a hostname, an IPv4 address, or an IPv6 address (not surrounded with brackets, as this is unnecessary since the port is specified separately). If a hostname resolves to an IPv4 address, an IPv4 socket will be used to send StatsD UDP packets, even if the hostname would also resolve to an IPv6 address.

The sample rate is a real number between 0 and 1 which defines the probability of sending a sample for any given event or timing measurement. This sample rate is sent with each sample to StatsD and used to multiply the value. For example, with a sample rate of 0.5, StatsD will multiply that counters value by 2 when flushing the metric to an upstream monitoring system (Graphite, Ganglia, etc.).

Some relatively high-frequency metrics have a default sample rate less than one. If you want to override the default sample rate for all metrics whose default sample rate is not specified in the Swift source, you may set *log_statsd_default_sample_rate* to a value less than one. This is NOT recommended (see next paragraph). A better way to reduce StatsD load is to adjust *log_statsd_sample_rate_factor* to a value less than one. The *log_statsd_sample_rate_factor* is multiplied to any sample rate (either the global default or one specified by the actual metric logging call in the Swift source) prior to handling. In other words, this one tunable can lower the frequency of all StatsD logging by a proportional amount.

To get the best data, start with the default *log_statsd_default_sample_rate* and *log_statsd_sample_rate_factor* values of 1 and only lower *log_statsd_sample_rate_factor* if needed. The *log_statsd_default_sample_rate* should not be used and remains for backward compatibility only.

The metric prefix will be prepended to every metric sent to the StatsD server For example, with:

```
log_statsd_metric_prefix = proxy01
```

the metric *proxy-server.errors* would be sent to StatsD as *proxy01.proxy-server.errors*. This is useful for differentiating different servers when sending statistics to a central StatsD server. If you run a local StatsD server per node, you could configure a per-node metrics prefix there and leave *log_statsd_metric_prefix* blank.

Note that metrics reported to StatsD are counters or timing data (which are sent in units of milliseconds). StatsD usually expands timing data out to min, max, avg, count, and 90th percentile per timing metric, but the details of this behavior will depend on the configuration of your StatsD server. Some important gauge metrics may still need to be collected using another method. For example, the *object-server.async_pendings* StatsD metric counts the generation of async_pendings in real-time, but will not tell you the current number of async_pending container updates on disk at any point in time.

Note also that the set of metrics collected, their names, and their semantics are not locked down and will change over time.

Metrics for *account-auditor*:

Metric Name	Description
account-auditor.errors	Count of audit runs (across all account databases) which caught an Exception.
account-auditor.passes	Count of individual account databases which passed audit.
account-auditor.failures	Count of individual account databases which failed audit.
account-auditor.timing	Timing data for individual account database audits.

Metrics for *account-reaper*:

Metric Name	Description
account-reaper.errors	Count of devices failing the mount check.
account-reaper.timing	Timing data for each reap_account() call.
account-reaper.return_codes.X	Count of HTTP return codes from various operations (e.g. object listing, container deletion, etc.). The value for X is the first digit of the return code (2 for 201, 4 for 404, etc.).
account-reaper.containers_failures	Count of failures to delete a container.
account-reaper.containers_deleted	Count of containers successfully deleted.
account-reaper.containers_remaining	Count of containers which failed to delete with zero successes.
account-reaper.containers_possibly_remaining	Count of containers which failed to delete with at least one success.
account-reaper.objects_failures	Count of failures to delete an object.
account-reaper.objects_deleted	Count of objects successfully deleted.
account-reaper.objects_remaining	Count of objects which failed to delete with zero successes.
account-reaper.objects_possibly_remaining	Count of objects which failed to delete with at least one success.

Metrics for *account-server* (Not Found is not considered an error and requests which increment *errors* are not included in the timing data):

Metric Name	Description
account-server.DELETE.errors.timing	Timing data for each DELETE request resulting in an error: bad request, not mounted, missing timestamp.
account-server.DELETE.timing	Timing data for each DELETE request not resulting in an error.
account-server.PUT.errors.timing	Timing data for each PUT request resulting in an error: bad request, not mounted, conflict, recently-deleted.
account-server.PUT.timing	Timing data for each PUT request not resulting in an error.
account-server.HEAD.errors.timing	Timing data for each HEAD request resulting in an error: bad request, not mounted.
account-server.HEAD.timing	Timing data for each HEAD request not resulting in an error.
account-server.GET.errors.timing	Timing data for each GET request resulting in an error: bad request, not mounted, bad delimiter, account listing limit too high, bad accept header.
account-server.GET.timing	Timing data for each GET request not resulting in an error.
account-server.REPLICATE.errors.timing	Timing data for each REPLICATE request resulting in an error: bad request, not mounted.
account-server.REPLICATE.timing	Timing data for each REPLICATE request not resulting in an error.
account-server.POST.errors.timing	Timing data for each POST request resulting in an error: bad request, bad or missing timestamp, not mounted.
account-server.POST.timing	Timing data for each POST request not resulting in an error.

Metrics for *account-replicator*:

Metric Name	Description
account-replicator.diffs	Count of syncs handled by sending differing rows.
account-replicator.diff_caps	Count of diffs operations which failed because max_diffs was hit.
account-replicator.no_changes	Count of accounts found to be in sync.
account-replicator.hashmatches	Count of accounts found to be in sync via hash comparison (*broker.merge_syncs* was called).
account-replicator.rsyncs	Count of completely missing accounts which were sent via rsync.
account-replicator.remote_merges	Count of syncs handled by sending entire database via rsync.
account-replicator.attempts	Count of database replication attempts.
account-replicator.failures	Count of database replication attempts which failed due to corruption (quarantined) or inability to read as well as attempts to individual nodes which failed.
account-replicator.removes.<device>	Count of databases on <device> deleted because the delete_timestamp was greater than the put_timestamp and the database had no rows or because it was successfully synced to other locations and doesnt belong here anymore.
account-replicator.successes	Count of replication attempts to an individual node which were successful.
account-replicator.timing	Timing data for each database replication attempt not resulting in a failure.

Metrics for *container-auditor*:

Metric Name	Description
container-auditor.errors	Incremented when an Exception is caught in an audit pass (only once per pass, max).
container-auditor.passes	Count of individual containers passing an audit.
container-auditor.failures	Count of individual containers failing an audit.
container-auditor.timing	Timing data for each container audit.

Metrics for *container-replicator*:

Metric Name	Description
container-replicator.diffs	Count of syncs handled by sending differing rows.
container-replicator.diff_caps	Count of diffs operations which failed because max_diffs was hit.
container-replicator.no_changes	Count of containers found to be in sync.
container-replicator.hashmatches	Count of containers found to be in sync via hash comparison (*broker.merge_syncs* was called).
container-replicator.rsyncs	Count of completely missing containers where were sent via rsync.
container-replicator.remote_merges	Count of syncs handled by sending entire database via rsync.
container-replicator.attempts	Count of database replication attempts.
container-replicator.failures	Count of database replication attempts which failed due to corruption (quarantined) or inability to read as well as attempts to individual nodes which failed.
container-replicator.removes.<device>	Count of databases deleted on <device> because the delete_timestamp was greater than the put_timestamp and the database had no rows or because it was successfully synced to other locations and doesnt belong here anymore.
container-replicator.successes	Count of replication attempts to an individual node which were successful.
container-replicator.timing	Timing data for each database replication attempt not resulting in a failure.

Metrics for *container-server* (Not Found is not considered an error and requests which increment *errors* are not included in the timing data):

Metric Name	Description
container-server.DELETE.errors.timing	Timing data for DELETE request errors: bad request, not mounted, missing timestamp, conflict.
container-server.DELETE.timing	Timing data for each DELETE request not resulting in an error.
container-server.PUT.errors.timing	Timing data for PUT request errors: bad request, missing timestamp, not mounted, conflict.
container-server.PUT.timing	Timing data for each PUT request not resulting in an error.
container-server.HEAD.errors.timing	Timing data for HEAD request errors: bad request, not mounted.
container-server.HEAD.timing	Timing data for each HEAD request not resulting in an error.
container-server.GET.errors.timing	Timing data for GET request errors: bad request, not mounted, parameters not utf8, bad accept header.
container-server.GET.timing	Timing data for each GET request not resulting in an error.
container-server.REPLICATE.errors.timing	Timing data for REPLICATE request errors: bad request, not mounted.
container-server.REPLICATE.timing	Timing data for each REPLICATE request not resulting in an error.
container-server.POST.errors.timing	Timing data for POST request errors: bad request, bad x-container-sync-to, not mounted.
container-server.POST.timing	Timing data for each POST request not resulting in an error.

Metrics for *container-sync*:

Metric Name	Description
container-sync.skips	Count of containers skipped because they dont have syncing enabled.
container-sync.failures	Count of failures syncing of individual containers.
container-sync.syncs	Count of individual containers synced successfully.
container-sync.deletes	Count of container database rows synced by deletion.
container-sync.deletes.timing	Timing data for each container database row synchronization via deletion.
container-sync.puts	Count of container database rows synced by Putting.
container-sync.puts.timing	Timing data for each container database row synchronization via Putting.

Metrics for *container-updater*:

Metric Name	Description
container-updater.successes	Count of containers which successfully updated their account.
container-updater.failures	Count of containers which failed to update their account.
container-updater.no_changes	Count of containers which didnt need to update their account.
container-updater.timing	Timing data for processing a container; only includes timing for containers which needed to update their accounts (i.e. successes and failures but not no_changes).

Metrics for *object-auditor*:

Metric Name	Description
object-auditor.quarantines	Count of objects failing audit and quarantined.
object-auditor.errors	Count of errors encountered while auditing objects.
object-auditor.timing	Timing data for each object audit (does not include any rate-limiting sleep time for max_files_per_second, but does include rate-limiting sleep time for max_bytes_per_second).

Metrics for *object-expirer*:

Metric Name	Description
object-expirer.objects	Count of objects expired.
object-expirer.errors	Count of errors encountered while attempting to expire an object.
object-expirer.timing	Timing data for each object expiration attempt, including ones resulting in an error.

Metrics for *object-reconstructor*:

Metric Name	Description
object-reconstructor.partition.delete.count.<device>	A count of partitions on <device> which were reconstructed and synced to another node because they didnt belong on this node. This metric is tracked per-device to allow for quiescence detection for object reconstruction activity on each device.
object-reconstructor.partition.delete.timing	Timing data for partitions reconstructed and synced to another node because they didnt belong on this node. This metric is not tracked per device.
object-reconstructor.partition.update.count.<device>	A count of partitions on <device> which were reconstructed and synced to another node, but also belong on this node. As with delete.count, this metric is tracked per-device.
object-reconstructor.partition.update.timing	Timing data for partitions reconstructed which also belong on this node. This metric is not tracked per-device.
object-reconstructor.suffix.hashes	Count of suffix directories whose hash (of filenames) was recalculated.
object-reconstructor.suffix.syncs	Count of suffix directories reconstructed with ssync.

Metrics for *object-replicator*:

Metric Name	Description
object-replicator.partition.delete.count.<device>	A count of partitions on <device> which were replicated to another node because they didnt belong on this node. This metric is tracked per-device to allow for quiescence detection for object replication activity on each device.
object-replicator.partition.delete.timing	Timing data for partitions replicated to another node because they didnt belong on this node. This metric is not tracked per device.
object-replicator.partition.update.count.<device>	A count of partitions on <device> which were replicated to another node, but also belong on this node. As with delete.count, this metric is tracked per-device.
object-replicator.partition.update.timing	Timing data for partitions replicated which also belong on this node. This metric is not tracked per-device.
object-replicator.suffix.hashes	Count of suffix directories whose hash (of filenames) was recalculated.
object-replicator.suffix.syncs	Count of suffix directories replicated with rsync.

Metrics for *object-server*:

Metric Name	Description
object-server.quarantines	Count of objects (files) found bad and moved to quarantine.
object-server.async_pendings	Count of container updates saved as async_pendings (may result from PUT or DELETE requests).
object-server.POST.errors.timing	Timing data for POST request errors: bad request, missing timestamp, delete-at in past, not mounted.
object-server.POST.timing	Timing data for each POST request not resulting in an error.
object-server.PUT.errors.timing	Timing data for PUT request errors: bad request, not mounted, missing timestamp, object creation constraint violation, delete-at in past.
object-server.PUT.timeouts	Count of object PUTs which exceeded max_upload_time.
object-server.PUT.timing	Timing data for each PUT request not resulting in an error.
object-server.PUT.<device>.timing	Timing data per kB transferred (ms/kB) for each non-zero-byte PUT request on each device. Monitoring problematic devices, higher is bad.
object-server.GET.errors.timing	Timing data for GET request errors: bad request, not mounted, header timestamps before the epoch, precondition failed. File errors resulting in a quarantine are not counted here.
object-server.GET.timing	Timing data for each GET request not resulting in an error. Includes requests which couldnt find the object (including disk errors resulting in file quarantine).
object-server.HEAD.errors.timing	Timing data for HEAD request errors: bad request, not mounted.
object-server.HEAD.timing	Timing data for each HEAD request not resulting in an error. Includes requests which couldnt find the object (including disk errors resulting in file quarantine).
object-server.DELETE.errors.timing	Timing data for DELETE request errors: bad request, missing timestamp, not mounted, precondition failed. Includes requests which couldnt find or match the object.
object-server.DELETE.timing	Timing data for each DELETE request not resulting in an error.
object-server.REPLICATE.errors.timing	Timing data for REPLICATE request errors: bad request, not mounted.
object-server.REPLICATE.timing	Timing data for each REPLICATE request not resulting in an error.

Metrics for *object-updater*:

Metric Name	Description
object-updater.errors	Count of drives not mounted or async_pending files with an unexpected name.
object-updater.timing	Timing data for object sweeps to flush async_pending container updates. Does not include object sweeps which did not find an existing async_pending storage directory.
object-updater.quarantines	Count of async_pending container updates which were corrupted and moved to quarantine.
object-updater.successes	Count of successful container updates.
object-updater.failures	Count of failed container updates.
object-updater.unlinks	Count of async_pending files unlinked. An async_pending file is unlinked either when it is successfully processed or when the replicator sees that there is a newer async_pending file for the same object.

Metrics for *proxy-server* (in the table, <*type*> is the proxy-server controller responsible for the request and will be one of account, container, or object):

Metric Name	Description
proxy-server.errors	Count of errors encountered while serving requests before the controller type is determined. Includes invalid Content-Length, errors finding the internal controller to handle the request, invalid utf8, and bad URLs.
proxy-server.<type>.handoff_count	Count of node hand-offs; only tracked if log_handoffs is set in the proxy-server config.
proxy-server.<type>.handoff_all_count	Count of times *only* hand-off locations were utilized; only tracked if log_handoffs is set in the proxy-server config.
proxy-server.<type>.client_timeouts	Count of client timeouts (client did not read within *client_timeout* seconds during a GET or did not supply data within *client_timeout* seconds during a PUT).
proxy-server.<type>.client_disconnects	Count of detected client disconnects during PUT operations (does NOT include caught Exceptions in the proxy-server which caused a client disconnect).

Metrics for *proxy-logging* middleware (in the table, <*type*> is either the proxy-server controller responsible for the request: account, container, object, or the string SOS if the request came from the Swift Origin Server middleware. The <*verb*> portion will be one of GET, HEAD, POST, PUT, DELETE, COPY, OPTIONS, or BAD_METHOD. The list of valid HTTP methods is configurable via the *log_statsd_valid_http_methods* config variable and the default setting yields the above behavior):

Metric Name	Description
proxy-server.<type>.<verb>.<status>.timing	Timing data for requests, start to finish. The <status> portion is the numeric HTTP status code for the request (e.g. 200 or 404).
proxy-server.<type>.GET.<status>.first-byte.timing	Timing data up to completion of sending the response headers (only for GET requests). <status> and <type> are as for the main timing metric.
proxy-server.<type>.<verb>.<status>.xfer	This counter metric is the sum of bytes transferred in (from clients) and out (to clients) for requests. The <type>, <verb>, and <status> portions of the metric are just like the main timing metric.

The *proxy-logging* middleware also groups these metrics by policy. The <*policy-index*> portion repre-

sents a policy index):

Metric Name	Description
proxy-server.object.policy.<policy-index>.<verb>.<status>.timing	Timing data for requests, aggregated by policy index.
proxy-server.object.policy.<policy-index>.GET.<status>.first-byte.timing	Timing data up to completion of sending the response headers, aggregated by policy index.
proxy-server.object.policy.<policy-index>.<verb>.<status>.xfer	Sum of bytes transferred in and out, aggregated by policy index.

Metrics for *tempauth* middleware (in the table, *<reseller_prefix>* represents the actual configured reseller_prefix or *NONE* if the reseller_prefix is the empty string):

Metric Name	Description
tempauth.<reseller_prefix>.unauthorized	Count of regular requests which were denied with HTTPUnauthorized.
tempauth.<reseller_prefix>.forbidden	Count of regular requests which were denied with HTTPForbidden.
tempauth.<reseller_prefix>.token_denied	Count of token requests which were denied.
tempauth.<reseller_prefix>.errors	Count of errors.

5.4.13 Debugging Tips and Tools

When a request is made to Swift, it is given a unique transaction id. This id should be in every log line that has to do with that request. This can be useful when looking at all the services that are hit by a single request.

If you need to know where a specific account, container or object is in the cluster, *swift-get-nodes* will show the location where each replica should be.

If you are looking at an object on the server and need more info, *swift-object-info* will display the account, container, replica locations and metadata of the object.

If you are looking at a container on the server and need more info, *swift-container-info* will display all the information like the account, container, replica locations and metadata of the container.

If you are looking at an account on the server and need more info, *swift-account-info* will display the account, replica locations and metadata of the account.

If you want to audit the data for an account, *swift-account-audit* can be used to crawl the account, checking that all containers and objects can be found.

5.4.14 Managing Services

Swift services are generally managed with `swift-init`. the general usage is `swift-init <service> <command>`, where service is the Swift service to manage (for example object, container, account, proxy) and command is one of:

Command	Description
start	Start the service
stop	Stop the service
restart	Restart the service
shutdown	Attempt to gracefully shutdown the service
reload	Attempt to gracefully restart the service
reload-seamless	Attempt to seamlessly restart the service

A graceful shutdown or reload will allow all server workers to finish any current requests before exiting. The parent server process exits immediately.

A seamless reload will make new configuration settings active, with no window where client requests fail due to there being no active listen socket. The parent server process will re-exec itself, retaining its existing PID. After the re-execed parent server process binds its listen sockets, the old listen sockets are closed and old server workers finish any current requests before exiting.

There is also a special case of `swift-init all <command>`, which will run the command for all swift services.

In cases where there are multiple configs for a service, a specific config can be managed with `swift-init <service>.<config> <command>`. For example, when a separate replication network is used, there might be `/etc/swift/object-server/public.conf` for the object server and `/etc/swift/object-server/replication.conf` for the replication services. In this case, the replication services could be restarted with `swift-init object-server.replication restart`.

5.4.15 Object Auditor

On system failures, the XFS file system can sometimes truncate files its trying to write and produce zero-byte files. The object-auditor will catch these problems but in the case of a system crash it would be advisable to run an extra, less rate limited sweep to check for these specific files. You can run this command as follows:

```
swift-object-auditor /path/to/object-server/config/file.conf once -z 1000
```

`-z` means to only check for zero-byte files at 1000 files per second.

At times it is useful to be able to run the object auditor on a specific device or set of devices. You can run the object-auditor as follows:

```
swift-object-auditor /path/to/object-server/config/file.conf once --
↪devices=sda,sdb
```

This will run the object auditor on only the sda and sdb devices. This param accepts a comma separated list of values.

5.4.16 Object Replicator

At times it is useful to be able to run the object replicator on a specific device or partition. You can run the object-replicator as follows:

```
swift-object-replicator /path/to/object-server/config/file.conf once --
↪devices=sda,sdb
```

This will run the object replicator on only the sda and sdb devices. You can likewise run that command with `--partitions`. Both params accept a comma separated list of values. If both are specified they will be ANDed together. These can only be run in once mode.

5.4.17 Swift Orphans

Swift Orphans are processes left over after a reload of a Swift server.

For example, when upgrading a proxy server you would probably finish with a `swift-init proxy-server reload` or `/etc/init.d/swift-proxy reload`. This kills the parent proxy server process and leaves the child processes running to finish processing whatever requests they might be handling at the time. It then starts up a new parent proxy server process and its children to handle new incoming requests. This allows zero-downtime upgrades with no impact to existing requests.

The orphaned child processes may take a while to exit, depending on the length of the requests they were handling. However, sometimes an old process can be hung up due to some bug or hardware issue. In these cases, these orphaned processes will hang around forever. `swift-orphans` can be used to find and kill these orphans.

`swift-orphans` with no arguments will just list the orphans it finds that were started more than 24 hours ago. You shouldnt really check for orphans until 24 hours after you perform a reload, as some requests can take a long time to process. `swift-orphans -k TERM` will send the SIG_TERM signal to the orphans processes, or you can `kill -TERM` the pids yourself if you prefer.

You can run `swift-orphans --help` for more options.

5.4.18 Swift Oldies

Swift Oldies are processes that have just been around for a long time. Theres nothing necessarily wrong with this, but it might indicate a hung process if you regularly upgrade and reload/restart services. You might have so many servers that you dont notice when a reload/restart fails; `swift-oldies` can help with this.

For example, if you upgraded and reloaded/restarted everything 2 days ago, and youve already cleaned up any orphans with `swift-orphans`, you can run `swift-oldies -a 48` to find any Swift processes still around that were started more than 2 days ago and then investigate them accordingly.

5.4.19 Custom Log Handlers

Swift supports setting up custom log handlers for services by specifying a comma-separated list of functions to invoke when logging is setup. It does so via the `log_custom_handlers` configuration option. Logger hooks invoked are passed the same arguments as Swifts get_logger function (as well as the getLogger and LogAdapter object):

Name	Description
conf	Configuration dict to read settings from
name	Name of the logger received
log_to_console	(optional) Write log messages to console on stderr
log_route	Route for the logging received
fmt	Override log format received
logger	The logging.getLogger object
adapted_logger	The LogAdapter object

A basic example that sets up a custom logger might look like the following:

```
def my_logger(conf, name, log_to_console, log_route, fmt, logger,
              adapted_logger):
    my_conf_opt = conf.get('some_custom_setting')
    my_handler = third_party_logstore_handler(my_conf_opt)
    logger.addHandler(my_handler)
```

See *Custom Logger Hooks* for sample use cases.

5.4.20 Securing OpenStack Swift

Please refer to the security guide at https://docs.openstack.org/security-guide and in particular the Object Storage section.

5.5 Dedicated replication network

5.5.1 Summary

Swifts replication process is essential for consistency and availability of data. By default, replication activity will use the same network interface as other cluster operations. However, if a replication interface is set in the ring for a node, that node will send replication traffic on its designated separate replication network interface. Replication traffic includes REPLICATE requests and rsync traffic.

To separate the cluster-internal replication traffic from client traffic, separate replication servers can be used. These replication servers are based on the standard storage servers, but they listen on the replication IP and only respond to REPLICATE requests. Storage servers can serve REPLICATE requests, so an operator can transition to using a separate replication network with no cluster downtime.

Replication IP and port information is stored in the ring on a per-node basis. These parameters will be used if they are present, but they are not required. If this information does not exist or is empty for a particular node, the nodes standard IP and port will be used for replication.

5.5.2 For SAIO replication

1. Create new script in ~/bin/ (for example: remakerings_new):

```
#!/bin/bash
set -e
cd /etc/swift
rm -f *.builder *.ring.gz backups/*.builder backups/*.ring.gz
swift-ring-builder object.builder create 10 3 1
swift-ring-builder object.builder add z1-127.0.0.1:6210R127.0.0.1:6250/
↪sdb1 1
swift-ring-builder object.builder add z2-127.0.0.1:6220R127.0.0.1:6260/
↪sdb2 1
swift-ring-builder object.builder add z3-127.0.0.1:6230R127.0.0.1:6270/
↪sdb3 1
swift-ring-builder object.builder add z4-127.0.0.1:6240R127.0.0.1:6280/
↪sdb4 1
swift-ring-builder object.builder rebalance
swift-ring-builder object-1.builder create 10 2 1
swift-ring-builder object-1.builder add z1-127.0.0.1:6210R127.0.0.1:6250/
↪sdb1 1
swift-ring-builder object-1.builder add z2-127.0.0.1:6220R127.0.0.1:6260/
↪sdb2 1
swift-ring-builder object-1.builder add z3-127.0.0.1:6230R127.0.0.1:6270/
↪sdb3 1
swift-ring-builder object-1.builder add z4-127.0.0.1:6240R127.0.0.1:6280/
↪sdb4 1
swift-ring-builder object-1.builder rebalance
swift-ring-builder object-2.builder create 10 6 1
swift-ring-builder object-2.builder add z1-127.0.0.1:6210R127.0.0.1:6250/
↪sdb1 1
swift-ring-builder object-2.builder add z1-127.0.0.1:6210R127.0.0.1:6250/
↪sdb5 1
swift-ring-builder object-2.builder add z2-127.0.0.1:6220R127.0.0.1:6260/
↪sdb2 1
swift-ring-builder object-2.builder add z2-127.0.0.1:6220R127.0.0.1:6260/
↪sdb6 1
swift-ring-builder object-2.builder add z3-127.0.0.1:6230R127.0.0.1:6270/
↪sdb3 1
swift-ring-builder object-2.builder add z3-127.0.0.1:6230R127.0.0.1:6270/
↪sdb7 1
swift-ring-builder object-2.builder add z4-127.0.0.1:6240R127.0.0.1:6280/
↪sdb4 1
swift-ring-builder object-2.builder add z4-127.0.0.1:6240R127.0.0.1:6280/
↪sdb8 1
swift-ring-builder object-2.builder rebalance
swift-ring-builder container.builder create 10 3 1
swift-ring-builder container.builder add z1-127.0.0.1:6211R127.0.0.1:6251/
↪sdb1 1
swift-ring-builder container.builder add z2-127.0.0.1:6221R127.0.0.1:6261/
↪sdb2 1
```

(continues on next page)

(continued from previous page)

```
swift-ring-builder container.builder add z3-127.0.0.1:6231R127.0.0.1:6271/
↪sdb3 1
swift-ring-builder container.builder add z4-127.0.0.1:6241R127.0.0.1:6281/
↪sdb4 1
swift-ring-builder container.builder rebalance
swift-ring-builder account.builder create 10 3 1
swift-ring-builder account.builder add z1-127.0.0.1:6212R127.0.0.1:6252/
↪sdb1 1
swift-ring-builder account.builder add z2-127.0.0.1:6222R127.0.0.1:6262/
↪sdb2 1
swift-ring-builder account.builder add z3-127.0.0.1:6232R127.0.0.1:6272/
↪sdb3 1
swift-ring-builder account.builder add z4-127.0.0.1:6242R127.0.0.1:6282/
↪sdb4 1
swift-ring-builder account.builder rebalance
```

Note: Syntax of adding device has been changed: `R<ip_replication>:<port_replication>` was added between `z<zone>-<ip>:<port>` and `/<device_name>_<meta> <weight>`. Added devices will use <ip_replication> and <port_replication> for replication activities.

2. Add next rows in `/etc/rsyncd.conf`:

```
[account6252]
max connections = 25
path = /srv/1/node/
read only = false
lock file = /var/lock/account6252.lock

[account6262]
max connections = 25
path = /srv/2/node/
read only = false
lock file = /var/lock/account6262.lock

[account6272]
max connections = 25
path = /srv/3/node/
read only = false
lock file = /var/lock/account6272.lock

[account6282]
max connections = 25
path = /srv/4/node/
read only = false
lock file = /var/lock/account6282.lock


[container6251]
```

(continues on next page)

(continued from previous page)

```
max connections = 25
path = /srv/1/node/
read only = false
lock file = /var/lock/container6251.lock

[container6261]
max connections = 25
path = /srv/2/node/
read only = false
lock file = /var/lock/container6261.lock

[container6271]
max connections = 25
path = /srv/3/node/
read only = false
lock file = /var/lock/container6271.lock

[container6281]
max connections = 25
path = /srv/4/node/
read only = false
lock file = /var/lock/container6281.lock


[object6250]
max connections = 25
path = /srv/1/node/
read only = false
lock file = /var/lock/object6250.lock

[object6260]
max connections = 25
path = /srv/2/node/
read only = false
lock file = /var/lock/object6260.lock

[object6270]
max connections = 25
path = /srv/3/node/
read only = false
lock file = /var/lock/object6270.lock

[object6280]
max connections = 25
path = /srv/4/node/
read only = false
lock file = /var/lock/object6280.lock
```

3. Restart rsync daemon:

```
service rsync restart
```

4. Update configuration files in directories:
   - /etc/swift/object-server(files: 1.conf, 2.conf, 3.conf, 4.conf)
   - /etc/swift/container-server(files: 1.conf, 2.conf, 3.conf, 4.conf)
   - /etc/swift/account-server(files: 1.conf, 2.conf, 3.conf, 4.conf)

   delete all configuration options in section `[<*>-replicator]`
5. Add configuration files for object-server, in `/etc/swift/object-server/`
   - 5.conf:

```
[DEFAULT]
devices = /srv/1/node
mount_check = false
disable_fallocate = true
bind_port = 6250
user = swift
log_facility = LOG_LOCAL2
recon_cache_path = /var/cache/swift

[pipeline:main]
pipeline = recon object-server

[app:object-server]
use = egg:swift#object
replication_server = True

[filter:recon]
use = egg:swift#recon

[object-replicator]
rsync_module = {replication_ip}::object{replication_port}
```

   - 6.conf:

```
[DEFAULT]
devices = /srv/2/node
mount_check = false
disable_fallocate = true
bind_port = 6260
user = swift
log_facility = LOG_LOCAL3
recon_cache_path = /var/cache/swift2

[pipeline:main]
pipeline = recon object-server

[app:object-server]
```

(continues on next page)

(continued from previous page)

```
use = egg:swift#object
replication_server = True

[filter:recon]
use = egg:swift#recon

[object-replicator]
rsync_module = {replication_ip}::object{replication_port}
```

- 7.conf:

```
[DEFAULT]
devices = /srv/3/node
mount_check = false
disable_fallocate = true
bind_port = 6270
user = swift
log_facility = LOG_LOCAL4
recon_cache_path = /var/cache/swift3

[pipeline:main]
pipeline = recon object-server

[app:object-server]
use = egg:swift#object
replication_server = True

[filter:recon]
use = egg:swift#recon

[object-replicator]
rsync_module = {replication_ip}::object{replication_port}
```

- 8.conf:

```
[DEFAULT]
devices = /srv/4/node
mount_check = false
disable_fallocate = true
bind_port = 6280
user = swift
log_facility = LOG_LOCAL5
recon_cache_path = /var/cache/swift4

[pipeline:main]
pipeline = recon object-server

[app:object-server]
use = egg:swift#object
replication_server = True
```

(continues on next page)

(continued from previous page)

```
[filter:recon]
use = egg:swift#recon

[object-replicator]
rsync_module = {replication_ip}::object{replication_port}
```

6. Add configuration files for container-server, in `/etc/swift/container-server/`

   - 5.conf:

```
[DEFAULT]
devices = /srv/1/node
mount_check = false
disable_fallocate = true
bind_port = 6251
user = swift
log_facility = LOG_LOCAL2
recon_cache_path = /var/cache/swift

[pipeline:main]
pipeline = recon container-server

[app:container-server]
use = egg:swift#container
replication_server = True

[filter:recon]
use = egg:swift#recon

[container-replicator]
rsync_module = {replication_ip}::container{replication_port}
```

   - 6.conf:

```
[DEFAULT]
devices = /srv/2/node
mount_check = false
disable_fallocate = true
bind_port = 6261
user = swift
log_facility = LOG_LOCAL3
recon_cache_path = /var/cache/swift2

[pipeline:main]
pipeline = recon container-server

[app:container-server]
use = egg:swift#container
replication_server = True
```

(continues on next page)

(continued from previous page)

```
[filter:recon]
use = egg:swift#recon

[container-replicator]
rsync_module = {replication_ip}::container{replication_port}
```

- 7.conf:

```
[DEFAULT]
devices = /srv/3/node
mount_check = false
disable_fallocate = true
bind_port = 6271
user = swift
log_facility = LOG_LOCAL4
recon_cache_path = /var/cache/swift3

[pipeline:main]
pipeline = recon container-server

[app:container-server]
use = egg:swift#container
replication_server = True

[filter:recon]
use = egg:swift#recon

[container-replicator]
rsync_module = {replication_ip}::container{replication_port}
```

- 8.conf:

```
[DEFAULT]
devices = /srv/4/node
mount_check = false
disable_fallocate = true
bind_port = 6281
user = swift
log_facility = LOG_LOCAL5
recon_cache_path = /var/cache/swift4

[pipeline:main]
pipeline = recon container-server

[app:container-server]
use = egg:swift#container
replication_server = True

[filter:recon]
```

(continues on next page)

(continued from previous page)

```
use = egg:swift#recon

[container-replicator]
rsync_module = {replication_ip}::container{replication_port}
```

7. Add configuration files for account-server, in `/etc/swift/account-server/`

   - 5.conf:

```
[DEFAULT]
devices = /srv/1/node
mount_check = false
disable_fallocate = true
bind_port = 6252
user = swift
log_facility = LOG_LOCAL2
recon_cache_path = /var/cache/swift

[pipeline:main]
pipeline = recon account-server

[app:account-server]
use = egg:swift#account
replication_server = True

[filter:recon]
use = egg:swift#recon

[account-replicator]
rsync_module = {replication_ip}::account{replication_port}
```

   - 6.conf:

```
[DEFAULT]
devices = /srv/2/node
mount_check = false
disable_fallocate = true
bind_port = 6262
user = swift
log_facility = LOG_LOCAL3
recon_cache_path = /var/cache/swift2

[pipeline:main]
pipeline = recon account-server

[app:account-server]
use = egg:swift#account
replication_server = True

[filter:recon]
```

(continues on next page)

(continued from previous page)

```
use = egg:swift#recon

[account-replicator]
rsync_module = {replication_ip}::account{replication_port}
```

- 7.conf:

```
[DEFAULT]
devices = /srv/3/node
mount_check = false
disable_fallocate = true
bind_port = 6272
user = swift
log_facility = LOG_LOCAL4
recon_cache_path = /var/cache/swift3

[pipeline:main]
pipeline = recon account-server

[app:account-server]
use = egg:swift#account
replication_server = True

[filter:recon]
use = egg:swift#recon

[account-replicator]
rsync_module = {replication_ip}::account{replication_port}
```

- 8.conf:

```
[DEFAULT]
devices = /srv/4/node
mount_check = false
disable_fallocate = true
bind_port = 6282
user = swift
log_facility = LOG_LOCAL5
recon_cache_path = /var/cache/swift4

[pipeline:main]
pipeline = recon account-server

[app:account-server]
use = egg:swift#account
replication_server = True

[filter:recon]
use = egg:swift#recon
```

(continues on next page)

(continued from previous page)

```
[account-replicator]
rsync_module = {replication_ip}::account{replication_port}
```

5.5.3 For a Multiple Server replication

1. Move configuration file.
   - Configuration file for object-server from /etc/swift/object-server.conf to /etc/swift/object-server/1.conf
   - Configuration file for container-server from /etc/swift/container-server.conf to /etc/swift/container-server/1.conf
   - Configuration file for account-server from /etc/swift/account-server.conf to /etc/swift/account-server/1.conf
2. Add changes in configuration files in directories:
   - /etc/swift/object-server(files: 1.conf)
   - /etc/swift/container-server(files: 1.conf)
   - /etc/swift/account-server(files: 1.conf)

   delete all configuration options in section [<*>-replicator]
3. Add configuration files for object-server, in /etc/swift/object-server/2.conf:

```
[DEFAULT]
bind_ip = $STORAGE_LOCAL_NET_IP
workers = 2

[pipeline:main]
pipeline = object-server

[app:object-server]
use = egg:swift#object
replication_server = True

[object-replicator]
```

4. Add configuration files for container-server, in /etc/swift/container-server/2.conf:

```
[DEFAULT]
bind_ip = $STORAGE_LOCAL_NET_IP
workers = 2

[pipeline:main]
pipeline = container-server

[app:container-server]
use = egg:swift#container
replication_server = True
```

(continues on next page)

(continued from previous page)

```
[container-replicator]
```

5. Add configuration files for account-server, in /etc/swift/account-server/2.conf:

```
[DEFAULT]
bind_ip = $STORAGE_LOCAL_NET_IP
workers = 2

[pipeline:main]
pipeline = account-server

[app:account-server]
use = egg:swift#account
replication_server = True

[account-replicator]
```

5.6 Logs

Swift has quite verbose logging, and the generated logs can be used for cluster monitoring, utilization calculations, audit records, and more. As an overview, Swifts logs are sent to syslog and organized by log level and syslog facility. All log lines related to the same request have the same transaction id. This page documents the log formats used in the system.

Note: By default, Swift will log full log lines. However, with the `log_max_line_length` setting and depending on your logging server software, lines may be truncated or shortened. With `log_max_line_length < 7`, the log line will be truncated. With `log_max_line_length >= 7`, the log line will be shortened: about half the max length followed by followed by the other half the max length. Unless you use exceptionally short values, you are unlikely to run across this with the following documented log lines, but you may see it with debugging and error log lines.

5.6.1 Proxy Logs

The proxy logs contain the record of all external API requests made to the proxy server. Swifts proxy servers log requests using a custom format designed to provide robust information and simple processing. It is possible to change this format with the `log_msg_template` config parameter. The default log format is:

```
{client_ip} {remote_addr} {end_time.datetime} {method} {path} {protocol}
    {status_int} {referer} {user_agent} {auth_token} {bytes_recvd}
    {bytes_sent} {client_etag} {transaction_id} {headers} {request_time}
    {source} {log_info} {start_time} {end_time} {policy_index}
```

Some keywords, signaled by the (anonymizable) flag, can be anonymized by using the transformer anonymized. The data are applied the hashing method of *log_anonymization_method* and an optional salt *log_anonymization_salt*.

Some keywords, signaled by the (timestamp) flag, can be converted to standard dates formats using the matching transformers: datetime, asctime or iso8601. Other transformers for timestamps are s, ms, us and ns for seconds, milliseconds, microseconds and nanoseconds. Pythons strftime directives can also be used as tranformers (a, A, b, B, c, d, H, I, j, m, M, p, S, U, w, W, x, X, y, Y, Z).

Example {client_ip.anonymized} {remote_addr.anonymized} {start_time.iso8601}
{end_time.H}:{end_time.M} {method} acc:{account} cnt:{container} obj:{object.anonymized}

Log Field	Value
client_ip	Swifts guess at the end-client IP, taken from various headers in the request. (anonymizable)
remote_addr	The IP address of the other end of the TCP connection. (anonymizable)
end_time	Timestamp of the request. (timestamp)
method	The HTTP verb in the request.
domain	The domain in the request. (anonymizable)
path	The path portion of the request. (anonymizable)
protocol	The transport protocol used (currently one of http or https).
status_int	The response code for the request.
referer	The value of the HTTP Referer header. (anonymizable)
user_agent	The value of the HTTP User-Agent header. (anonymizable)
auth_token	The value of the auth token. This may be truncated or otherwise obscured.
bytes_recvd	The number of bytes read from the client for this request.
bytes_sent	The number of bytes sent to the client in the body of the response. This is how many bytes were yielded to the WSGI server.
client_etag	The etag header value given by the client. (anonymizable)
transaction_id	The transaction id of the request.
headers	The headers given in the request. (anonymizable)
request_time	The duration of the request.
source	The source of the request. This may be set for requests that are generated in order to fulfill client requests, e.g. bulk uploads.
log_info	Various info that may be useful for diagnostics, e.g. the value of any x-delete-at header.
start_time	High-resolution timestamp from the start of the request. (timestamp)
end_time	High-resolution timestamp from the end of the request. (timestamp)
ttfb	Duration between the request and the first bytes is sent.
policy_index	The value of the storage policy index.
account	The account part extracted from the path of the request. (anonymizable)
container	The container part extracted from the path of the request. (anonymizable)
object	The object part extracted from the path of the request. (anonymizable)
pid	PID of the process emitting the log line.
wire_status_int	The status sent to the client, which may be different than the logged response code if there was an error during the body of the request or a disconnect.

In one log line, all of the above fields are space-separated and url-encoded. If any value is empty, it will be logged as a -. This allows for simple parsing by splitting each line on whitespace. New values

may be placed at the end of the log line from time to time, but the order of the existing values will not change. Swift log processing utilities should look for the first N fields they require (e.g. in Python using something like `log_line.split()[:14]` to get up through the transaction id).

Note: Some log fields (like the request path) are already url quoted, so the logged value will be double-quoted. For example, if a client uploads an object name with a `:` in it, it will be url-quoted as `%3A`. The log module will then quote this value as `%253A`.

Swift Source

The `source` value in the proxy logs is used to identify the originator of a request in the system. For example, if the client initiates a bulk upload, the proxy server may end up doing many requests. The initial bulk upload request will be logged as normal, but all of the internal child requests will have a source value indicating they came from the bulk functionality.

Logged Source Value	Originator of the Request
FP	*FormPost*
SLO	*Static Large Objects*
SW	*StaticWeb*
TU	*TempURL*
BD	*Bulk Operations (Delete and Archive Auto Extraction)* (delete)
EA	*Bulk Operations (Delete and Archive Auto Extraction)* (extract)
AQ	*Account Quotas*
CQ	*Container Quotas*
CS	*Container Sync Middleware*
TA	*TempAuth*
DLO	*Dynamic Large Objects*
LE	*List Endpoints*
KS	*KeystoneAuth*
RL	*Rate Limiting*
RO	*Read Only*
VW	*Versioned Writes*
SSC	*Server Side Copy*
SYM	*Symlink*
SH	*Container Sharding*
S3	*AWS S3 Api*
OV	*Object Versioning*
EQ	*Etag Quoter*

5.6.2 Storage Node Logs

Swifts account, container, and object server processes each log requests that they receive, if they have been configured to do so with the `log_requests` config parameter (which defaults to true). The format for these log lines is:

```
remote_addr - - [datetime] "request_method request_path" status_int
    content_length "referer" "transaction_id" "user_agent" request_time
    additional_info server_pid policy_index
```

Log Field	Value
remote_addr	The IP address of the other end of the TCP connection.
datetime	Timestamp of the request, in day/month/year:hour:minute:second +0000 format.
request_method	The HTTP verb in the request.
request_path	The path portion of the request.
status_int	The response code for the request.
content_length	The value of the Content-Length header in the response.
referer	The value of the HTTP Referer header.
transaction_id	The transaction id of the request.
user_agent	The value of the HTTP User-Agent header. Swift services report a user-agent string of the service name followed by the process ID, such as "`proxy-server <pid of the proxy>`" or "`object-updater <pid of the object updater>`".
request_time	The time between request received and response started. **Note**: This includes transfer time on PUT, but not GET.
additional_info	Additional useful information.
server_pid	The process id of the server
policy_index	The value of the storage policy index.

5.7 Swift Ops Runbook

This document contains operational procedures that Hewlett Packard Enterprise (HPE) uses to operate and monitor the Swift system within the HPE Helion Public Cloud. This document is an excerpt of a larger product-specific handbook. As such, the material may appear incomplete. The suggestions and recommendations made in this document are for our particular environment, and may not be suitable for your environment or situation. We make no representations concerning the accuracy, adequacy, completeness or suitability of the information, suggestions or recommendations. This document are provided

for reference only. We are not responsible for your use of any information, suggestions or recommendations contained herein.

5.7.1 Identifying issues and resolutions

Is the system up?

If you have a report that Swift is down, perform the following basic checks:

1. Run swift functional tests.
2. From a server in your data center, use `curl` to check `/healthcheck` (see below).
3. If you have a monitoring system, check your monitoring system.
4. Check your hardware load balancers infrastructure.
5. Run swift-recon on a proxy node.

Functional tests usage

We would recommend that you set up the functional tests to run against your production system. Run regularly this can be a useful tool to validate that the system is configured correctly. In addition, it can provide early warning about failures in your system (if the functional tests stop working, user applications will also probably stop working).

A script for running the function tests is located in `swift/.functests`.

External monitoring

We use pingdom.com to monitor the external Swift API. We suggest the following:

- Do a GET on `/healthcheck`
- Create a container, make it public (`x-container-read: .r*,.rlistings`), create a small file in the container; do a GET on the object

Diagnose: General approach

- Look at service status in your monitoring system.
- In addition to system monitoring tools and issue logging by users, swift errors will often result in log entries (see *Diagnose: Interpreting messages in /var/log/swift/ files*).
- Look at any logs your deployment tool produces.
- Log files should be reviewed for error signatures (see below) that may point to a known issue, or root cause issues reported by the diagnostics tools, prior to escalation.

Dependencies

The Swift software is dependent on overall system health. Operating system level issues with network connectivity, domain name resolution, user management, hardware and system configuration and capacity in terms of memory and free disk space, may result is secondary Swift issues. System level issues should be resolved prior to diagnosis of swift issues.

Diagnose: Swift-dispersion-report

The swift-dispersion-report is a useful tool to gauge the general health of the system. Configure the `swift-dispersion` report to cover at a minimum every disk drive in your system (usually 1% coverage). See *Dispersion Report* for details of how to configure and use the dispersion reporting tool.

The `swift-dispersion-report` tool can take a long time to run, especially if any servers are down. We suggest you run it regularly (e.g., in a cron job) and save the results. This makes it easy to refer to the last report without having to wait for a long-running command to complete.

Diagnose: Is system responding to `/healthcheck`?

When you want to establish if a swift endpoint is running, run `curl -k` against `https://$ENDPOINT/healthcheck`.

Diagnose: Interpreting messages in `/var/log/swift/` files

Note: In the Hewlett Packard Enterprise Helion Public Cloud we send logs to `proxy.log` (proxy-server logs), `server.log` (object-server, account-server, container-server logs), `background.log` (all other servers [object-replicator, etc]).

The following table lists known issues:

Logfile	Signature	Issue	Steps to take
/var/log/syslog	kernel: [] sd . [csbu:sd] Sense Key: Medium Error	Suggests disk surface issues	Run `swift-drive-audit` on the target node to check for disk errors, repair disk errors
/var/log/syslog	kernel: [] sd . [csbu:sd] Sense Key: Hardware Error	Suggests storage hardware issues	Run diagnostics on the target node to check for disk failures, replace failed disks
/var/log/syslog	kernel: [] . I/O error, dev sd. ,sector .		Run diagnostics on the target node to check for disk errors
/var/log/syslog	pound: NULL get_thr_arg	Multiple threads woke up	Noise, safe to ignore
/var/log/swift/proxy.log	. ERROR . ConnectionTimeout .	A storage node is not responding in a timely fashion	Check if node is down, not running Swift, unconfigured, storage off-line or for network issues between the proxy and non responding node
/var/log/swift/proxy.log	proxy-server . HTTP/1.0 500 .	A proxy server has reported an internal server error	Examine the logs for any errors at the time the error was reported to attempt to understand the cause of the error.
/var/log/swift/server.log	. ERROR . ConnectionTimeout .	A storage server is not responding in a timely fashion	Check if node is down, not running Swift, unconfigured, storage off-line or for network issues between the server and non responding node
/var/log/swift/server.log	. ERROR . Remote I/O error: /srv/node/disk.	A storage device is not responding as expected	Run `swift-drive-audit` and check the filesystem named in the error for corruption (unmount & xfs_repair). Check if the filesystem is mounted and working.
/var/log/swift/background	object-server ERROR container update failed . Connection refused	A container server node could not be contacted	Check if node is down, not running Swift, unconfigured, storage off-line or for network issues between the server and non responding node
/var/log/swift/background	object-updater ERROR with remote . ConnectionTimeout	The remote container server is busy	If the container is very large, some errors updating it can be expected. However, this error can also occur if

Diagnose: Parted reports the backup GPT table is corrupt

- If a GPT table is broken, a message like the following should be observed when the following command is run:

```
$ sudo parted -l
```

```
Error: The backup GPT table is corrupt, but the primary appears OK,
so that will be used.

OK/Cancel?
```

To fix, go to *Fix broken GPT table (broken disk partition)*

Diagnose: Drives diagnostic reports a FS label is not acceptable

If diagnostics reports something like FS label: obj001dsk011 is not acceptable, it indicates that a partition has a valid disk label, but an invalid filesystem label. In such cases proceed as follows:

1. Verify that the disk labels are correct:

```
$ FS=/dev/sd#1

$ sudo parted -l | grep object
```

2. If partition labels are inconsistent then, resolve the disk label issues before proceeding:

```
$ sudo parted -s ${FS} name ${PART_NO} ${PART_NAME} #Partition Label
$ # PART_NO is 1 for object disks and 3 for OS disks
$ # PART_NAME follows the convention seen in "sudo parted -l | grep object
↪"
```

3. If the Filesystem label is missing then create it with care:

```
$ sudo xfs_admin -l ${FS} #Filesystem label (12 Char limit)

$ # Check for the existence of a FS label

$ OBJNO=<3 Length Object No.>

$ # I.E OBJNO for sw-stbaz3-object0007 would be 007

$ DISKNO=<3 Length Disk No.>

$ # I.E DISKNO for /dev/sdb would be 001, /dev/sdc would be 002 etc.

$ sudo xfs_admin -L "obj${OBJNO}dsk${DISKNO}" ${FS}

$ # Create a FS Label
```

Diagnose: Failed LUNs

Note: The HPE Helion Public Cloud uses direct attach SmartArray controllers/drives. The information here is specific to that environment. The hpacucli utility mentioned here may be called hpssacli in your environment.

The `swift_diagnostics` mount checks may return a warning that a LUN has failed, typically accompanied by DriveAudit check failures and device errors.

Such cases are typically caused by a drive failure, and if drive check also reports a failed status for the underlying drive, then follow the procedure to replace the disk.

Otherwise the lun can be re-enabled as follows:

1. Generate a hpssacli diagnostic report. This report allows the DC team to troubleshoot potential cabling or hardware issues so it is imperative that you run it immediately when troubleshooting a failed LUN. You will come back later and grep this file for more details, but just generate it for now.

```
$ sudo hpssacli controller all diag file=/tmp/hpacu.diag ris=on xml=off␣
↪zip=off
```

Export the following variables using the below instructions before proceeding further.

1. Print a list of logical drives and their numbers and take note of the failed drives number and array value (example output: array A logicaldrive 1 would be exported as LDRIVE=1):

```
$ sudo hpssacli controller slot=1 ld all show
```

2. Export the number of the logical drive that was retrieved from the previous command into the LDRIVE variable:

```
$ export LDRIVE=<LogicalDriveNumber>
```

3. Print the array value and Port:Box:Bay for all drives and take note of the Port:Box:Bay for the failed drive (example output: array A physicaldrive 2C:1:1 would be exported as PBOX=2C:1:1). Match the array value of this output with the array value obtained from the previous command to be sure you are working on the same drive. Also, the array value usually matches the device name (For example, /dev/sdc in the case of array c), but we will run a different command to be sure we are operating on the correct device.

```
$ sudo hpssacli controller slot=1 pd all show
```

Note: Sometimes a LUN may appear to be failed as it is not and cannot be mounted but the hpssacli/parted commands may show no problems with the LUNS/drives. In this case, the filesystem may be corrupt and may be necessary to run `sudo xfs_check /dev/sd[a-l][1-2]` to see if there is an xfs issue. The results of running this command may require that `xfs_repair` is run.

1. Export the Port:Box:Bay for the failed drive into the PBOX variable:

```
$ export PBOX=<Port:Box:Bay>
```

2. Print the physical device information and take note of the Disk Name (example output: Disk Name: /dev/sdk would be exported as DEV=/dev/sdk):

```
$ sudo hpssacli controller slot=1 ld ${LDRIVE} show detail | grep -i
↪"Disk Name"
```

3. Export the device name variable from the preceding command (example: /dev/sdk):

```
$ export DEV=<Device>
```

4. Export the filesystem variable. Disks that are split between the operating system and data storage, typically sda and sdb, should only have repairs done on their data filesystem, usually /dev/sda2 and /dev/sdb2, Other data only disks have just one partition on the device, so the filesystem will be 1. In any case you should verify the data filesystem by running `df -h | grep /srv/node` and using the listed data filesystem for the device in question as the export. For example: /dev/sdk1.

```
$ export FS=<Filesystem>
```

5. Verify the LUN is failed, and the device is not:

```
$ sudo hpssacli controller slot=1 ld all show
$ sudo hpssacli controller slot=1 pd all show
$ sudo hpssacli controller slot=1 ld ${LDRIVE} show detail
$ sudo hpssacli controller slot=1 pd ${PBOX} show detail
```

6. Stop the swift and rsync service:

```
$ sudo service rsync stop
$ sudo swift-init shutdown all
```

7. Unmount the problem drive, fix the LUN and the filesystem:

```
$ sudo umount ${FS}
```

8. If umount fails, you should run lsof search for the mountpoint and kill any lingering processes before repeating the unpount:

```
$ sudo hpacucli controller slot=1 ld ${LDRIVE} modify reenable
$ sudo xfs_repair ${FS}
```

9. If the `xfs_repair` complains about possible journal data, use the `xfs_repair -L` option to zeroise the journal log.

10. Once complete test-mount the filesystem, and tidy up its lost and found area.

```
$ sudo mount ${FS} /mnt
$ sudo rm -rf /mnt/lost+found/
$ sudo umount /mnt
```

11. Mount the filesystem and restart swift and rsync.

12. Run the following to determine if a DC ticket is needed to check the cables on the node:

```
$ grep -y media.exchanged /tmp/hpacu.diag
$ grep -y hot.plug.count /tmp/hpacu.diag
```

13. If the output reports any non 0x00 values, it suggests that the cables should be checked. For example, log a DC ticket to check the sas cables between the drive and the expander.

Diagnose: Slow disk devices

Note: collectl is an open-source performance gathering/analysis tool.

If the diagnostics report a message such as `sda: drive is slow`, you should log onto the node and run the following command (remove `-c 1` option to continuously monitor the data):

```
$ /usr/bin/collectl -s D -c 1
waiting for 1 second sample...
# DISK STATISTICS (/sec)
#          <---------reads---------><---------writes---------><--------
↪averages--------> Pct
#Name       KBytes Merged  IOs Size  KBytes Merged  IOs Size  RWSize  QLen ␣
↪Wait SvcTim Util
sdb            204      0   33    6      43      0    4   11       6     1 ␣
↪ 7      6   23
sda             84      0   13    6     108     21    6   18      10     1 ␣
↪ 7      7   13
sdc            100      0   16    6       0      0    0    0       6     1 ␣
↪ 7      6    9
sdd            140      0   22    6      22      0    2   11       6     1 ␣
↪ 9      9   22
sde             76      0   12    6     255      0   52    5       5     1 ␣
↪ 2      1   10
sdf            276      0   44    6       0      0    0    0       6     1 ␣
↪11      8   38
sdg            112      0   17    7      18      0    2    9       6     1 ␣
↪ 7      7   13
sdh           3552      0   73   49       0      0    0    0      48     1 ␣
↪ 9      8   62
sdi              72      0   12    6       0      0    0    0       6     1 ␣
↪ 8      8   10
sdj             112      0   17    7      22      0    2   11       7     1 ␣
↪10      9   18
sdk             120      0   19    6      21      0    2   11       6     1 ␣
↪ 8      8   16
sdl             144      0   22    7      18      0    2    9       6     1 ␣
↪ 9      7   18
dm-0              0      0    0    0       0      0    0    0       0     0 ␣
↪ 0      0    0
dm-1              0      0    0    0      60      0   15    4       4     0 ␣
↪ 0      0    0
```

(continues on next page)

(continued from previous page)

```
dm-2              0       0    0    0       48       0   12     4        4      0  ␣
↪ 0       0    0
dm-3              0       0    0    0        0       0    0     0        0      0  ␣
↪ 0       0    0
dm-4              0       0    0    0        0       0    0     0        0      0  ␣
↪ 0       0    0
dm-5              0       0    0    0        0       0    0     0        0      0  ␣
↪ 0       0    0
```

Look at the `Wait` and `SvcTime` values. It is not normal for these values to exceed 50msec. This is known to impact customer performance (upload/download). For a controller problem, many/all drives will show long wait and service times. A reboot may correct the problem; otherwise hardware replacement is needed.

Another way to look at the data is as follows:

```
$ /opt/hp/syseng/disk-anal.pl -d
Disk: sda  Wait: 54580 371  65  25  12   6   6   0   1   2   0  46
Disk: sdb  Wait: 54532 374  96  36  16   7   4   1   0   2   0  46
Disk: sdc  Wait: 54345 554 105  29  15   4   7   1   4   4   0  46
Disk: sdd  Wait: 54175 553 254  31  20  11   6   6   2   2   1  53
Disk: sde  Wait: 54923  66  56  15   8   7   7   0   1   0   2  29
Disk: sdf  Wait: 50952 941 565 403 426 366 442 447 338  99  38  97
Disk: sdg  Wait: 50711 689 808 562 642 675 696 185  43  14   7  82
Disk: sdh  Wait: 51018 668 688 483 575 542 692 275  55  22   9  87
Disk: sdi  Wait: 51012 1011 849 672 568 240 344 280  38  13   6  81
Disk: sdj  Wait: 50724 743 770 586 662 509 684 283  46  17  11  79
Disk: sdk  Wait: 50886 700 585 517 633 511 729 352  89  23   8  81
Disk: sdl  Wait: 50106 617 794 553 604 504 532 501 288 234 165 216
Disk: sda  Time: 55040  22  16   6   1   1  13   0   0   0   3  12

Disk: sdb  Time: 55014  41  19   8   3   1   8   0   0   0   3  17
Disk: sdc  Time: 55032  23  14   8   9   2   6   1   0   0   0  19
Disk: sdd  Time: 55022  29  17  12   6   2  11   0   0   0   1  14
Disk: sde  Time: 55018  34  15  11  12   1   9   0   0   0   2  12
Disk: sdf  Time: 54809 250  45   7   1   0   0   0   0   0   1   1
Disk: sdg  Time: 55070  36   6   2   0   0   0   0   0   0   0   0
Disk: sdh  Time: 55079  33   2   0   0   0   0   0   0   0   0   0
Disk: sdi  Time: 55074  28   7   2   0   0   2   0   0   0   0   1
Disk: sdj  Time: 55067  35  10   0   1   0   0   0   0   0   0   1
Disk: sdk  Time: 55068  31  10   3   0   0   1   0   0   0   0   1
Disk: sdl  Time: 54905 130  61   7   3   4   1   0   0   0   0   3
```

This shows the historical distribution of the wait and service times over a day. This is how you read it:

- sda did 54580 operations with a short wait time, 371 operations with a longer wait time and 65 with an even longer wait time.
- sdl did 50106 operations with a short wait time, but as you can see many took longer.

There is a clear pattern that sdf to sdl have a problem. Actually, sda to sde would more normally have lots of zeros in their data. But maybe this is a busy system. In this example it is worth changing the controller

as the individual drives may be ok.

After the controller is changed, use collectl -s D as described above to see if the problem has cleared. disk-anal.pl will continue to show historical data. You can look at recent data as follows. It only looks at data from 13:15 to 14:15. As you can see, this is a relatively clean system (few if any long wait or service times):

```
$ /opt/hp/syseng/disk-anal.pl -d -t 13:15-14:15
Disk: sda  Wait:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdb  Wait:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdc  Wait:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdd  Wait:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sde  Wait:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdf  Wait:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdg  Wait:  3594   6   0   0   0   0   0   0   0   0   0   0
Disk: sdh  Wait:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdi  Wait:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdj  Wait:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdk  Wait:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdl  Wait:  3599   1   0   0   0   0   0   0   0   0   0   0
Disk: sda  Time:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdb  Time:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdc  Time:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdd  Time:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sde  Time:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdf  Time:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdg  Time:  3594   6   0   0   0   0   0   0   0   0   0   0
Disk: sdh  Time:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdi  Time:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdj  Time:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdk  Time:  3600   0   0   0   0   0   0   0   0   0   0   0
Disk: sdl  Time:  3599   1   0   0   0   0   0   0   0   0   0   0
```

For long wait times, where the service time appears normal is to check the logical drive cache status. While the cache may be enabled, it can be disabled on a per-drive basis.

Diagnose: Slow network link - Measuring network performance

Network faults can cause performance between Swift nodes to degrade. Testing with `netperf` is recommended. Other methods (such as copying large files) may also work, but can produce inconclusive results.

Install `netperf` on all systems if not already installed. Check that the UFW rules for its control port are in place. However, there are no pre-opened ports for netperfs data connection. Pick a port number. In this example, 12866 is used because it is one higher than netperfs default control port number, 12865. If you get very strange results including zero values, you may not have gotten the data port opened in UFW at the target or may have gotten the netperf command-line wrong.

Pick a `source` and `target` node. The source is often a proxy node and the target is often an object node. Using the same source proxy you can test communication to different object nodes in different AZs to identity possible bottlenecks.

Running tests

1. Prepare the `target` node as follows:

```
$ sudo iptables -I INPUT -p tcp -j ACCEPT
```

   Or, do:

```
$ sudo ufw allow 12866/tcp
```

2. On the `source` node, run the following command to check throughput. Note the double-dash before the -P option. The command takes 10 seconds to complete. The `target` node is 192.168.245.5.

```
$ netperf -H 192.168.245.5 -- -P 12866
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 12866 AF_INET to
<redacted>.72.4 (<redacted>.72.4) port 12866 AF_INET : demo
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec
87380  16384  16384    10.02     923.69
```

3. On the `source` node, run the following command to check latency:

```
$ netperf -H 192.168.245.5 -t TCP_RR -- -P 12866
MIGRATED TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 12866
AF_INET to <redacted>.72.4 (<redacted>.72.4) port 12866 AF_INET : demo
: first burst 0
Local  Remote Socket   Size    Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec
16384  87380  1        1       10.00    11753.37
16384  87380
```

Expected results

Faults will show up as differences between different pairs of nodes. However, for reference, here are some expected numbers:

- For throughput, proxy to proxy, expect ~9300 Mbit/sec (proxies have a 10Ge link).
- For throughout, proxy to object, expect ~920 Mbit/sec (at time of writing this, object nodes have a 1Ge link).
- For throughput, object to object, expect ~920 Mbit/sec.
- For latency (all types), expect ~11000 transactions/sec.

Diagnose: Remapping sectors experiencing UREs

1. Find the bad sector, device, and filesystem in `kern.log`.
2. Set the environment variables SEC, DEV & FS, for example:

```
$ SEC=2930954256
$ DEV=/dev/sdi
$ FS=/dev/sdi1
```

3. Verify that the sector is bad:

```
$ sudo dd if=${DEV} of=/dev/null bs=512 count=1 skip=${SEC}
```

4. If the sector is bad this command will output an input/output error:

```
dd: reading `/dev/sdi`: Input/output error
0+0 records in
0+0 records out
```

5. Prevent chef from attempting to re-mount the filesystem while the repair is in progress:

```
$ sudo mv /etc/chef/client.pem /etc/chef/xx-client.xx-pem
```

6. Stop the swift and rsync service:

```
$ sudo service rsync stop
$ sudo swift-init shutdown all
```

7. Unmount the problem drive:

```
$ sudo umount ${FS}
```

8. Overwrite/remap the bad sector:

```
$ sudo dd_rescue -d -A -m8b -s ${SEC}b ${DEV} ${DEV}
```

9. This command should report an input/output error the first time it is run. Run the command a second time, if it successfully remapped the bad sector it should not report an input/output error.
10. Verify the sector is now readable:

```
$ sudo dd if=${DEV} of=/dev/null bs=512 count=1 skip=${SEC}
```

11. If the sector is now readable this command should not report an input/output error.
12. If more than one problem sector is listed, set the SEC environment variable to the next sector in the list:

```
$ SEC=123456789
```

13. Repeat from step 8.
14. Repair the filesystem:

```
$ sudo xfs_repair ${FS}
```

15. If `xfs_repair` reports that the filesystem has valuable filesystem changes:

```
$ sudo xfs_repair ${FS}
Phase 1 - find and verify superblock...
Phase 2 - using internal log
        - zero log...
ERROR: The filesystem has valuable metadata changes in a log which
needs to be replayed.
Mount the filesystem to replay the log, and unmount it before
re-running xfs_repair.
If you are unable to mount the filesystem, then use the -L option to
destroy the log and attempt a repair. Note that destroying the log may
cause corruption -- please attempt a mount of the filesystem before
doing this.
```

16. You should attempt to mount the filesystem, and clear the lost+found area:

```
$ sudo mount $FS /mnt
$ sudo rm -rf /mnt/lost+found/*
$ sudo umount /mnt
```

17. If the filesystem fails to mount then you will need to use the `xfs_repair -L` option to force log zeroing. Repeat step 11.

18. If `xfs_repair` reports that an additional input/output error has been encountered, get the sector details as follows:

```
$ sudo grep "I/O error" /var/log/kern.log | grep sector | tail -1
```

19. If new input/output error is reported then set the SEC environment variable to the problem sector number:

```
$ SEC=234567890
```

20. Repeat from step 8

21. Remount the filesystem and restart swift and rsync.

    - If all UREs in the kern.log have been fixed and you are still unable to have xfs_repair disk, it is possible that the UREs have corrupted the filesystem or possibly destroyed the drive altogether. In this case, the first step is to re-format the filesystem and if this fails, get the disk replaced.

Diagnose: High system latency

Note: The latency measurements described here are specific to the HPE Helion Public Cloud.

- A bad NIC on a proxy server. However, as explained above, this usually causes the peak to rise, but average should remain near normal parameters. A quick fix is to shutdown the proxy.
- A stuck memcache server. Accepts connections, but then will not respond. Expect to see timeout messages in `/var/log/proxy.log` (port 11211). Swift Diags will also report this as a failed node/port. A quick fix is to shutdown the proxy server.
- A bad/broken object server can also cause problems if the accounts used by the monitor program happen to live on the bad object server.
- A general network problem within the data canter. Compare the results with the Pingdom monitors to see if they also have a problem.

Diagnose: Interface reports errors

Should a network interface on a Swift node begin reporting network errors, it may well indicate a cable, switch, or network issue.

Get an overview of the interface with:

```
$ sudo ifconfig eth{n}
$ sudo ethtool eth{n}
```

The `Link Detected:` indicator will read `yes` if the nic is cabled.

Establish the adapter type with:

```
$ sudo ethtool  -i eth{n}
```

Gather the interface statistics with:

```
$ sudo ethtool  -S eth{n}
```

If the nick supports self test, this can be performed with:

```
$ sudo ethtool  -t eth{n}
```

Self tests should read `PASS` if the nic is operating correctly.

Nic module drivers can be re-initialised by carefully removing and re-installing the modules (this avoids rebooting the server). For example, mellanox drivers use a two part driver mlx4_en and mlx4_core. To reload these you must carefully remove the mlx4_en (ethernet) then the mlx4_core modules, and reinstall them in the reverse order.

As the interface will be disabled while the modules are unloaded, you must be very careful not to lock yourself out so it may be better to script this.

Diagnose: Hung swift object replicator

A replicator reports in its log that remaining time exceeds 100 hours. This may indicate that the swift `object-replicator` is stuck and not making progress. Another useful way to check this is with the swift-recon -r command on a swift proxy server:

```
$ sudo swift-recon -r
===============================================================================

--> Starting reconnaissance on 384 hosts
===============================================================================
[2013-07-17 12:56:19] Checking on replication
[replication_time] low: 2, high: 80, avg: 28.8, total: 11037, Failed: 0.0%,␣
↪no_result: 0, reported: 383
Oldest completion was 2013-06-12 22:46:50 (12 days ago) by 192.168.245.3:6200.
Most recent completion was 2013-07-17 12:56:19 (5 seconds ago) by 192.168.245.
↪5:6200.
===============================================================================
```

The `Oldest completion` line in this example indicates that the object-replicator on swift object server 192.168.245.3 has not completed the replication cycle in 12 days. This replicator is stuck. The object replicator cycle is generally less than 1 hour. Though an replicator cycle of 15-20 hours can occur if nodes are added to the system and a new ring has been deployed.

You can further check if the object replicator is stuck by logging on the object server and checking the object replicator progress with the following command:

```
$ sudo grep object-rep /var/log/swift/background.log | grep -e "Starting␣
↪object replication" -e "Object replication complete" -e "partitions rep"
Jul 16 06:25:46 192.168.245.4 object-replicator 15344/16450 (93.28%)␣
↪partitions replicated in 69018.48s (0.22/sec, 22h remaining)
Jul 16 06:30:46 192.168.245.4object-replicator 15344/16450 (93.28%)␣
↪partitions replicated in 69318.58s (0.22/sec, 22h remaining)
Jul 16 06:35:46 192.168.245.4 object-replicator 15344/16450 (93.28%)␣
↪partitions replicated in 69618.63s (0.22/sec, 23h remaining)
Jul 16 06:40:46 192.168.245.4 object-replicator 15344/16450 (93.28%)␣
↪partitions replicated in 69918.73s (0.22/sec, 23h remaining)
Jul 16 06:45:46 192.168.245.4 object-replicator 15348/16450 (93.30%)␣
↪partitions replicated in 70218.75s (0.22/sec, 24h remaining)
Jul 16 06:50:47 192.168.245.4object-replicator 15348/16450 (93.30%)␣
↪partitions replicated in 70518.85s (0.22/sec, 24h remaining)
Jul 16 06:55:47 192.168.245.4 object-replicator 15348/16450 (93.30%)␣
↪partitions replicated in 70818.95s (0.22/sec, 25h remaining)
Jul 16 07:00:47 192.168.245.4 object-replicator 15348/16450 (93.30%)␣
↪partitions replicated in 71119.05s (0.22/sec, 25h remaining)
Jul 16 07:05:47 192.168.245.4 object-replicator 15348/16450 (93.30%)␣
↪partitions replicated in 71419.15s (0.21/sec, 26h remaining)
Jul 16 07:10:47 192.168.245.4object-replicator 15348/16450 (93.30%)␣
↪partitions replicated in 71719.25s (0.21/sec, 26h remaining)
Jul 16 07:15:47 192.168.245.4 object-replicator 15348/16450 (93.30%)␣
↪partitions replicated in 72019.27s (0.21/sec, 27h remaining)
```

(continues on next page)

(continued from previous page)

```
Jul 16 07:20:47 192.168.245.4object-replicator 15348/16450 (93.30%)␣
↪partitions replicated in 72319.37s (0.21/sec, 27h remaining)
Jul 16 07:25:47 192.168.245.4 object-replicator 15348/16450 (93.30%)␣
↪partitions replicated in 72619.47s (0.21/sec, 28h remaining)
Jul 16 07:30:47 192.168.245.4 object-replicator 15348/16450 (93.30%)␣
↪partitions replicated in 72919.56s (0.21/sec, 28h remaining)
Jul 16 07:35:47 192.168.245.4 object-replicator 15348/16450 (93.30%)␣
↪partitions replicated in 73219.67s (0.21/sec, 29h remaining)
Jul 16 07:40:47 192.168.245.4 object-replicator 15348/16450 (93.30%)␣
↪partitions replicated in 73519.76s (0.21/sec, 29h remaining)
```

The above status is output every 5 minutes to `/var/log/swift/background.log`.

Note: The remaining time is increasing as time goes on, normally the time remaining should be decreasing. Also note the partition number. For example, 15344 remains the same for several status lines. Eventually the object replicator detects the hang and attempts to make progress by killing the problem thread. The replicator then progresses to the next partition but quite often it again gets stuck on the same partition.

One of the reasons for the object replicator hanging like this is filesystem corruption on the drive. The following is a typical log entry of a corrupted filesystem detected by the object replicator:

```
$ sudo bzgrep "Remote I/O error" /var/log/swift/background.log* |grep srv | -␣
↪tail -1
Jul 12 03:33:30 192.168.245.4 object-replicator STDOUT: ERROR:root:Error␣
↪hashing suffix#012Traceback (most recent call last):#012 File
"/usr/lib/python2.7/dist-packages/swift/obj/replicator.py", line 199, in get_
↪hashes#012 hashes[suffix] = hash_suffix(suffix_dir,
reclaim_age)#012 File "/usr/lib/python2.7/dist-packages/swift/obj/replicator.
↪py", line 84, in hash_suffix#012 path_contents =
sorted(os.listdir(path))#012OSError: [Errno 121] Remote I/O error: '/srv/node/
↪disk4/objects/1643763/b51'
```

An `ls` of the problem file or directory usually shows something like the following:

```
$ ls -l /srv/node/disk4/objects/1643763/b51
ls: cannot access /srv/node/disk4/objects/1643763/b51: Remote I/O error
```

If no entry with `Remote I/O error` occurs in the `background.log` it is not possible to determine why the object-replicator is hung. It may be that the `Remote I/O error` entry is older than 7 days and so has been rotated out of the logs. In this scenario it may be best to simply restart the object-replicator.

1. Stop the object-replicator:

   ```
   # sudo swift-init object-replicator stop
   ```

2. Make sure the object replicator has stopped, if it has hung, the stop command will not stop the hung process:

```
# ps auxww | - grep swift-object-replicator
```

3. If the previous ps shows the object-replicator is still running, kill the process:

```
# kill -9 <pid-of-swift-object-replicator>
```

4. Start the object-replicator:

```
# sudo swift-init object-replicator start
```

If the above grep did find an `Remote I/O error` then it may be possible to repair the problem filesystem.

1. Stop swift and rsync:

```
# sudo swift-init all shutdown
# sudo service rsync stop
```

2. Make sure all swift process have stopped:

```
# ps auxww | grep swift | grep python
```

3. Kill any swift processes still running.
4. Unmount the problem filesystem:

```
# sudo umount /srv/node/disk4
```

5. Repair the filesystem:

```
# sudo xfs_repair -P /dev/sde1
```

6. If the `xfs_repair` fails then it may be necessary to re-format the filesystem. See *Procedure: Fix broken XFS filesystem*. If the `xfs_repair` is successful, re-enable chef using the following command and replication should commence again.

Diagnose: High CPU load

The CPU load average on an object server, as shown with the uptime command, is typically under 10 when the server is lightly-moderately loaded:

```
$ uptime
07:59:26 up 99 days,  5:57,  1 user,  load average: 8.59, 8.39, 8.32
```

During times of increased activity, due to user transactions or object replication, the CPU load average can increase to to around 30.

However, sometimes the CPU load average can increase significantly. The following is an example of an object server that has extremely high CPU load:

```
$ uptime
07:44:02 up 18:22,  1 user,  load average: 407.12, 406.36, 404.59
```

Further issues and resolutions

Note: The urgency levels in each **Action** column indicates whether or not it is required to take immediate action, or if the problem can be worked on during business hours.

Scenario	Description	Action
`/healthcheck` latency is high.	The `/healthcheck` test does not tax the proxy very much so any drop in value is probably related to network issues, rather than the proxies being very busy. A very slow proxy might impact the average number, but it would need to be very slow to shift the number that much.	Check networks. Do a `curl https://<ip-address>:<port>/healthcheck` where `ip-address` is individual proxy IP address. Repeat this for every proxy server to see if you can pin point the problem. Urgency: If there are other indications that your system is slow, you should treat this as an urgent problem.
Swift process is not running.	You can use `swift-init` status to check if swift processes are running on any given server.	Run this command: `$ sudo swift-init all␣ ↪start` Examine messages in the swift log files to see if there are any error messages related to any of the swift processes since the time you ran the `swift-init` command. Take any corrective actions that seem necessary. Urgency: If this only affects one server, and you have more than one, identifying and fixing the problem can wait until business hours. If this same problem affects many servers, then you need to take corrective action immediately.
ntpd is not running.	NTP is not running.	Configure and start NTP. Urgency: For proxy servers, this is vital.
Host clock is not syncd to an NTP server.	Node time settings does not match NTP server time. This may take some time to sync after a reboot.	Assuming NTP is configured and running, you have to wait until the times sync.
A swift process has hundreds, to thousands of open file descriptors.	May happen to any of the swift processes. Known to have happened with a `rsyslod` restart and where `/tmp` was hanging.	Restart the swift processes on the affected node: `$ sudo swift-init all␣ ↪reload` **Urgency:** If known performance problem: Immediate If system seems fine: Medium
A swift process is not owned by the swift user.	If the UID of the swift user has changed, then the processes might not be owned by that UID.	Urgency: If this only affects one server, and you have more than one, identifying and fixing the problem can wait until business hours. If this same problem affects many servers, then you

5.7.2 Software configuration procedures

Fix broken GPT table (broken disk partition)

- If a GPT table is broken, a message like the following should be observed when the command

```
$ sudo parted -l
```

- is run.

```
...
Error: The backup GPT table is corrupt, but the primary appears OK, so␣
↪that will
be used.
OK/Cancel?
```

1. To fix this, firstly install the `gdisk` program to fix this:

```
$ sudo aptitude install gdisk
```

2. Run `gdisk` for the particular drive with the damaged partition:

3. On the command prompt, type `r` (recovery and transformation options), followed by `d` (use main GPT header) , `v` (verify disk) and finally `w` (write table to disk and exit). Will also need to enter `Y` when prompted in order to confirm actions.

```
Command (? for help): r

Recovery/transformation command (? for help): d

Recovery/transformation command (? for help): v

Caution: The CRC for the backup partition table is invalid. This table may
be corrupt. This program will automatically create a new backup partition
table when you save your partitions.

Caution: Partition 1 doesn't begin on a 8-sector boundary. This may
result in degraded performance on some modern (2009 and later) hard disks.

Caution: Partition 2 doesn't begin on a 8-sector boundary. This may
result in degraded performance on some modern (2009 and later) hard disks.

Caution: Partition 3 doesn't begin on a 8-sector boundary. This may
result in degraded performance on some modern (2009 and later) hard disks.

Identified 1 problems!

Recovery/transformation command (? for help): w

Final checks complete. About to write GPT data. THIS WILL OVERWRITE␣
↪EXISTING
PARTITIONS!!
```

(continues on next page)

(continued from previous page)

```
Do you want to proceed, possibly destroying your data? (Y/N): Y

OK; writing new GUID partition table (GPT).
The operation has completed successfully.
```

4. Running the command:

```
$ sudo parted /dev/sd#
```

5. Should now show that the partition is recovered and healthy again.

6. Finally, uninstall `gdisk` from the node:

```
$ sudo aptitude remove gdisk
```

Procedure: Fix broken XFS filesystem

1. A filesystem may be corrupt or broken if the following output is observed when checking its label:

```
$ sudo xfs_admin -l /dev/sd#
cache_node_purge: refcount was 1, not zero (node=0x25d5ee0)
xfs_admin: cannot read root inode (117)
cache_node_purge: refcount was 1, not zero (node=0x25d92b0)
xfs_admin: cannot read realtime bitmap inode (117)
bad sb magic # 0 in AG 1
failed to read label in AG 1
```

2. Run the following commands to remove the broken/corrupt filesystem and replace. (This example uses the filesystem `/dev/sdb2`) Firstly need to replace the partition:

```
$ sudo parted
GNU Parted 2.3
Using /dev/sda
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted) select /dev/sdb
Using /dev/sdb
(parted) p
Model: HP LOGICAL VOLUME (scsi)
Disk /dev/sdb: 2000GB
Sector size (logical/physical): 512B/512B
Partition Table: gpt

Number  Start   End     Size    File system  Name   Flags
1      17.4kB  1024MB  1024MB  ext3                boot
2      1024MB  1751GB  1750GB  xfs          sw-aw2az1-object045-disk1
3      1751GB  2000GB  249GB                       lvm

(parted) rm 2
(parted) mkpart primary 2 -1
```

(continues on next page)

(continued from previous page)

```
Warning: You requested a partition from 2000kB to 2000GB.
The closest location we can manage is 1024MB to 1751GB.
Is this still acceptable to you?
Yes/No? Yes
Warning: The resulting partition is not properly aligned for best␣
↪performance.
Ignore/Cancel? Ignore
(parted) p
Model: HP LOGICAL VOLUME (scsi)
Disk /dev/sdb: 2000GB
Sector size (logical/physical): 512B/512B
Partition Table: gpt

Number  Start   End     Size    File system  Name     Flags
1       17.4kB  1024MB  1024MB  ext3                  boot
2       1024MB  1751GB  1750GB  xfs          primary
3       1751GB  2000GB  249GB                         lvm

(parted) quit
```

3. Next step is to scrub the filesystem and format:

```
$ sudo dd if=/dev/zero of=/dev/sdb2 bs=$((1024*1024)) count=1
1+0 records in
1+0 records out
1048576 bytes (1.0 MB) copied, 0.00480617 s, 218 MB/s
$ sudo /sbin/mkfs.xfs -f -i size=1024 /dev/sdb2
meta-data=/dev/sdb2              isize=1024   agcount=4, agsize=106811524␣
↪blks
         =                       sectsz=512   attr=2, projid32bit=0
data     =                       bsize=4096   blocks=427246093, imaxpct=5
         =                       sunit=0      swidth=0 blks
naming   =version 2              bsize=4096   ascii-ci=0
log      =internal log           bsize=4096   blocks=208616, version=2
         =                       sectsz=512   sunit=0 blks, lazy-count=1
realtime =none                   extsz=4096   blocks=0, rtextents=0
```

4. You should now label and mount your filesystem.

5. Can now check to see if the filesystem is mounted using the command:

```
$ mount
```

Procedure: Checking if an account is okay

Note: `swift-direct` is only available in the HPE Helion Public Cloud. Use `swiftly` as an alternate (or use `swift-get-nodes` as explained here).

You must know the tenant/project ID. You can check if the account is okay as follows from a proxy.

```
$ sudo -u swift  /opt/hp/swift/bin/swift-direct show AUTH_<project-id>
```

The response will either be similar to a swift list of the account containers, or an error indicating that the resource could not be found.

Alternatively, you can use `swift-get-nodes` to find the account database files. Run the following on a proxy:

```
$ sudo swift-get-nodes /etc/swift/account.ring.gz  AUTH_<project-id>
```

The response will print curl/ssh commands that will list the replicated account databases. Use the indicated `curl` or `ssh` commands to check the status and existence of the account.

Procedure: Getting swift account stats

Note: `swift-direct` is specific to the HPE Helion Public Cloud. Go look at `swifty` for an alternate or use `swift-get-nodes` as explained in *Procedure: Checking if an account is okay*.

This procedure describes how you determine the swift usage for a given swift account, that is the number of containers, number of objects and total bytes used. To do this you will need the project ID.

Log onto one of the swift proxy servers.

Use swift-direct to show this accounts usage:

```
$ sudo -u swift /opt/hp/swift/bin/swift-direct show AUTH_<project-id>
Status: 200
      Content-Length: 0
      Accept-Ranges: bytes
      X-Timestamp: 1379698586.88364
      X-Account-Bytes-Used: 67440225625994
      X-Account-Container-Count: 1
      Content-Type: text/plain; charset=utf-8
      X-Account-Object-Count: 8436776
      Status: 200
      name: my_container  count: 8436776  bytes: 67440225625994
```

This account has 1 container. That container has 8436776 objects. The total bytes used is 67440225625994.

Procedure: Revive a deleted account

Swift accounts are normally not recreated. If a tenant/project is deleted, the account can then be deleted. If the user wishes to use Swift again, the normal process is to create a new tenant/project and hence a new Swift account.

However, if the Swift account is deleted, but the tenant/project is not deleted from Keystone, the user can no longer access the account. This is because the account is marked deleted in Swift. You can revive the account as described in this process.

Note: The containers and objects in the old account cannot be listed anymore. In addition, if the Account Reaper process has not finished reaping the containers and objects in the old account, these are effectively orphaned and it is virtually impossible to find and delete them to free up disk space.

The solution is to delete the account database files and re-create the account as follows:

1. You must know the tenant/project ID. The account name is AUTH_<project-id>. In this example, the tenant/project is `4ebe3039674d4864a11fe0864ae4d905` so the Swift account name is `AUTH_4ebe3039674d4864a11fe0864ae4d905`.

2. Use `swift-get-nodes` to locate the accounts database files (on three servers). The output has been truncated so we can focus on the import pieces of data:

```
$ sudo swift-get-nodes /etc/swift/account.ring.gz AUTH_
↪4ebe3039674d4864a11fe0864ae4d905
...
curl -I -XHEAD "http://192.168.245.5:6202/disk1/3934/AUTH_
↪4ebe3039674d4864a11fe0864ae4d905"
curl -I -XHEAD "http://192.168.245.3:6202/disk0/3934/AUTH_
↪4ebe3039674d4864a11fe0864ae4d905"
curl -I -XHEAD "http://192.168.245.4:6202/disk1/3934/AUTH_
↪4ebe3039674d4864a11fe0864ae4d905"
...
Use your own device location of servers:
such as "export DEVICE=/srv/node"
ssh 192.168.245.5 "ls -lah ${DEVICE:-/srv/node*}/disk1/accounts/3934/052/
↪f5ecf8b40de3e1b0adb0dbe576874052"
ssh 192.168.245.3 "ls -lah ${DEVICE:-/srv/node*}/disk0/accounts/3934/052/
↪f5ecf8b40de3e1b0adb0dbe576874052"
ssh 192.168.245.4 "ls -lah ${DEVICE:-/srv/node*}/disk1/accounts/3934/052/
↪f5ecf8b40de3e1b0adb0dbe576874052"
...
note: `/srv/node*` is used as default value of `devices`, the real value␣
↪is set in the config file on each storage node.
```

3. Before proceeding check that the account is really deleted by using curl. Execute the commands printed by `swift-get-nodes`. For example:

```
$ curl -I -XHEAD "http://192.168.245.5:6202/disk1/3934/AUTH_
↪4ebe3039674d4864a11fe0864ae4d905"
HTTP/1.1 404 Not Found
```

(continues on next page)

(continued from previous page)

```
Content-Length: 0
Content-Type: text/html; charset=utf-8
```

Repeat for the other two servers (192.168.245.3 and 192.168.245.4). A `404 Not Found` indicates that the account is deleted (or never existed).

If you get a `204 No Content` response, do **not** proceed.

4. Use the ssh commands printed by `swift-get-nodes` to check if database files exist. For example:

```
$  ssh 192.168.245.5 "ls -lah ${DEVICE:-/srv/node*}/disk1/accounts/3934/
↪052/f5ecf8b40de3e1b0adb0dbe576874052"
total 20K
drwxr-xr-x 2 swift swift 110 Mar  9 10:22 .
drwxr-xr-x 3 swift swift  45 Mar  9 10:18 ..
-rw------- 1 swift swift 17K Mar  9 10:22␣
↪f5ecf8b40de3e1b0adb0dbe576874052.db
-rw-r--r-- 1 swift swift   0 Mar  9 10:22␣
↪f5ecf8b40de3e1b0adb0dbe576874052.db.pending
-rwxr-xr-x 1 swift swift   0 Mar  9 10:18 .lock
```

Repeat for the other two servers (192.168.245.3 and 192.168.245.4).

If no files exist, no further action is needed.

5. Stop Swift processes on all nodes listed by `swift-get-nodes` (In this example, that is 192.168.245.3, 192.168.245.4 and 192.168.245.5).

6. We recommend you make backup copies of the database files.

7. Delete the database files. For example:

```
$ ssh 192.168.245.5
$ cd /srv/node/disk1/accounts/3934/052/f5ecf8b40de3e1b0adb0dbe576874052
$ sudo rm *
```

Repeat for the other two servers (192.168.245.3 and 192.168.245.4).

8. Restart Swift on all three servers

At this stage, the account is fully deleted. If you enable the auto-create option, the next time the user attempts to access the account, the account will be created. You may also use swiftly to recreate the account.

Procedure: Temporarily stop load balancers from directing traffic to a proxy server

You can stop the load balancers sending requests to a proxy server as follows. This can be useful when a proxy is misbehaving but you need Swift running to help diagnose the problem. By removing from the load balancers, customers are not impacted by the misbehaving proxy.

1. Ensure that in /etc/swift/proxy-server.conf the `disable_path` variable is set to `/etc/swift/disabled-by-file`.

2. Log onto the proxy node.

3. Shut down Swift as follows:

```
$ sudo swift-init proxy shutdown
```

Note: Shutdown, not stop.

4. Create the `/etc/swift/disabled-by-file` file. For example:

```
$ sudo touch /etc/swift/disabled-by-file
```

5. Optional, restart Swift:

```
$ sudo swift-init proxy start
```

It works because the healthcheck middleware looks for /etc/swift/disabled-by-file. If it exists, the middleware will return 503/error instead of 200/OK. This means the load balancer should stop sending traffic to the proxy.

Procedure: Ad-Hoc disk performance test

You can get an idea whether a disk drive is performing as follows:

```
$ sudo dd bs=1M count=256 if=/dev/zero conv=fdatasync of=/srv/node/disk11/
↪remember-to-delete-this-later
```

You can expect ~600MB/sec. If you get a low number, repeat many times as Swift itself may also read or write to the disk, hence giving a lower number.

5.7.3 Server maintenance

General assumptions

- It is assumed that anyone attempting to replace hardware components will have already read and understood the appropriate maintenance and service guides.
- It is assumed that where servers need to be taken off-line for hardware replacement, that this will be done in series, bringing the server back on-line before taking the next off-line.
- It is assumed that the operations directed procedure will be used for identifying hardware for replacement.

Assessing the health of swift

You can run the swift-recon tool on a Swift proxy node to get a quick check of how Swift is doing. Please note that the numbers below are necessarily somewhat subjective. Sometimes parameters for which we say low values are good will have pretty high values for a time. Often if you wait a while things get better.

For example:

```
$ sudo swift-recon -rla
===============================================================================
[2012-03-10 12:57:21] Checking async pendings on 384 hosts...
Async stats: low: 0, high: 1, avg: 0, total: 1
===============================================================================

[2012-03-10 12:57:22] Checking replication times on 384 hosts...
[Replication Times] shortest: 1.4113877813, longest: 36.8293570836, avg: 4.
↪86278064749
===============================================================================

[2012-03-10 12:57:22] Checking load avg's on 384 hosts...
[5m load average] lowest: 2.22, highest: 9.5, avg: 4.59578125
[15m load average] lowest: 2.36, highest: 9.45, avg: 4.62622395833
[1m load average] lowest: 1.84, highest: 9.57, avg: 4.5696875
===============================================================================
```

In the example above we ask for information on replication times (-r), load averages (-l) and async pendings (-a). This is a healthy Swift system. Rules-of-thumb for good recon output are:

- Nodes that respond are up and running Swift. If all nodes respond, that is a good sign. But some nodes may time out. For example:

```
-> [http://<redacted>.29:6200/recon/load:] <urlopen error [Errno 111]␣
↪ECONNREFUSED>
-> [http://<redacted>.31:6200/recon/load:] <urlopen error timed out>
```

- That could be okay or could require investigation.
- Low values (say < 10 for high and average) for async pendings are good. Higher values occur when disks are down and/or when the system is heavily loaded. Many simultaneous PUTs to the same container can drive async pendings up. This may be normal, and may resolve itself after a while. If it persists, one way to track down the problem is to find a node with high async pendings (with `swift-recon -av | sort -n -k4`), then check its Swift logs, Often async pendings are high because a node cannot write to a container on another node. Often this is because the node or disk is offline or bad. This may be okay if we know about it.
- Low values for replication times are good. These values rise when new rings are pushed, and when nodes and devices are brought back on line.
- Our high load average values are typically in the 9-15 range. If they are a lot bigger it is worth having a look at the systems pushing the average up. Run `swift-recon -av` to get the individual averages. To sort the entries with the highest at the end, run `swift-recon -av | sort -n -k4`.

For comparison here is the recon output for the same system above when two entire racks of Swift are down:

```
[2012-03-10 16:56:33] Checking async pendings on 384 hosts...
-> http://<redacted>.22:6200/recon/async: <urlopen error timed out>
-> http://<redacted>.18:6200/recon/async: <urlopen error timed out>
-> http://<redacted>.16:6200/recon/async: <urlopen error timed out>
-> http://<redacted>.13:6200/recon/async: <urlopen error timed out>
-> http://<redacted>.30:6200/recon/async: <urlopen error timed out>
-> http://<redacted>.6:6200/recon/async: <urlopen error timed out>
.........
-> http://<redacted>.5:6200/recon/async: <urlopen error timed out>
-> http://<redacted>.15:6200/recon/async: <urlopen error timed out>
-> http://<redacted>.9:6200/recon/async: <urlopen error timed out>
-> http://<redacted>.27:6200/recon/async: <urlopen error timed out>
-> http://<redacted>.4:6200/recon/async: <urlopen error timed out>
-> http://<redacted>.8:6200/recon/async: <urlopen error timed out>
Async stats: low: 243, high: 659, avg: 413, total: 132275
================================================================================
[2012-03-10 16:57:48] Checking replication times on 384 hosts...
-> http://<redacted>.22:6200/recon/replication: <urlopen error timed out>
-> http://<redacted>.18:6200/recon/replication: <urlopen error timed out>
-> http://<redacted>.16:6200/recon/replication: <urlopen error timed out>
-> http://<redacted>.13:6200/recon/replication: <urlopen error timed out>
-> http://<redacted>.30:6200/recon/replication: <urlopen error timed out>
-> http://<redacted>.6:6200/recon/replication: <urlopen error timed out>
............
-> http://<redacted>.5:6200/recon/replication: <urlopen error timed out>
-> http://<redacted>.15:6200/recon/replication: <urlopen error timed out>
-> http://<redacted>.9:6200/recon/replication: <urlopen error timed out>
-> http://<redacted>.27:6200/recon/replication: <urlopen error timed out>
-> http://<redacted>.4:6200/recon/replication: <urlopen error timed out>
-> http://<redacted>.8:6200/recon/replication: <urlopen error timed out>
[Replication Times] shortest: 1.38144306739, longest: 112.620954418, avg: 10.
↪285
9475361
================================================================================
[2012-03-10 16:59:03] Checking load avg's on 384 hosts...
-> http://<redacted>.22:6200/recon/load: <urlopen error timed out>
-> http://<redacted>.18:6200/recon/load: <urlopen error timed out>
-> http://<redacted>.16:6200/recon/load: <urlopen error timed out>
-> http://<redacted>.13:6200/recon/load: <urlopen error timed out>
-> http://<redacted>.30:6200/recon/load: <urlopen error timed out>
-> http://<redacted>.6:6200/recon/load: <urlopen error timed out>
............
-> http://<redacted>.15:6200/recon/load: <urlopen error timed out>
-> http://<redacted>.9:6200/recon/load: <urlopen error timed out>
-> http://<redacted>.27:6200/recon/load: <urlopen error timed out>
-> http://<redacted>.4:6200/recon/load: <urlopen error timed out>
-> http://<redacted>.8:6200/recon/load: <urlopen error timed out>
[5m load average] lowest: 1.71, highest: 4.91, avg: 2.486375
[15m load average] lowest: 1.79, highest: 5.04, avg: 2.506125
[1m load average] lowest: 1.46, highest: 4.55, avg: 2.4929375
```

(continues on next page)

(continued from previous page)

```
==============================================================================
```

Note: The replication times and load averages are within reasonable parameters, even with 80 object stores down. Async pendings, however is quite high. This is due to the fact that the containers on the servers which are down cannot be updated. When those servers come back up, async pendings should drop. If async pendings were at this level without an explanation, we have a problem.

Recon examples

Here is an example of noting and tracking down a problem with recon.

Running reccon shows some async pendings:

```
$ ssh -q <redacted>.132.7 sudo swift-recon -alr
==============================================================================
[2012-03-14 17:25:55] Checking async pendings on 384 hosts...
Async stats: low: 0, high: 23, avg: 8, total: 3356
==============================================================================
[2012-03-14 17:25:55] Checking replication times on 384 hosts...
[Replication Times] shortest: 1.49303831657, longest: 39.6982825994, avg: 4.
↪2418222066
==============================================================================
[2012-03-14 17:25:56] Checking load avg's on 384 hosts...
[5m load average] lowest: 2.35, highest: 8.88, avg: 4.45911458333
[15m load average] lowest: 2.41, highest: 9.11, avg: 4.504765625
[1m load average] lowest: 1.95, highest: 8.56, avg: 4.40588541667
└
↪==============================================================================
```

Why? Running recon again with -av swift (not shown here) tells us that the node with the highest (23) is <redacted>.72.61. Looking at the log files on <redacted>.72.61 we see:

```
$ sudo tail -f /var/log/swift/background.log | - grep -i ERROR
Mar 14 17:28:06 <redacted> container-replicator ERROR Remote drive not mounted
{'zone': 5, 'weight': 1952.0, 'ip': '<redacted>.204.119', 'id': 5481, 'meta':
↪'', 'device': 'disk6', 'port': 6201}
Mar 14 17:28:06 <redacted> container-replicator ERROR Remote drive not mounted
{'zone': 5, 'weight': 1952.0, 'ip': '<redacted>.204.119', 'id': 5481, 'meta':
↪'', 'device': 'disk6', 'port': 6201}
Mar 14 17:28:09 <redacted> container-replicator ERROR Remote drive not mounted
{'zone': 5, 'weight': 1952.0, 'ip': '<redacted>.204.20', 'id': 2311, 'meta': '
↪', 'device': 'disk5', 'port': 6201}
Mar 14 17:28:11 <redacted> container-replicator ERROR Remote drive not mounted
{'zone': 5, 'weight': 1952.0, 'ip': '<redacted>.204.20', 'id': 2311, 'meta': '
↪', 'device': 'disk5', 'port': 6201}
Mar 14 17:28:13 <redacted> container-replicator ERROR Remote drive not mounted
{'zone': 5, 'weight': 1952.0, 'ip': '<redacted>.204.119', 'id': 5481, 'meta':
↪'', 'device': 'disk6', 'port': 6201}
```

(continues on next page)

(continued from previous page)

```
Mar 14 17:28:13 <redacted> container-replicator ERROR Remote drive not mounted
{'zone': 5, 'weight': 1952.0, 'ip': '<redacted>.204.119', 'id': 5481, 'meta':
↪'', 'device': 'disk6', 'port': 6201}
Mar 14 17:28:15 <redacted> container-replicator ERROR Remote drive not mounted
{'zone': 5, 'weight': 1952.0, 'ip': '<redacted>.204.20', 'id': 2311, 'meta': '
↪', 'device': 'disk5', 'port': 6201}
Mar 14 17:28:15 <redacted> container-replicator ERROR Remote drive not mounted
{'zone': 5, 'weight': 1952.0, 'ip': '<redacted>.204.20', 'id': 2311, 'meta': '
↪', 'device': 'disk5', 'port': 6201}
Mar 14 17:28:19 <redacted> container-replicator ERROR Remote drive not mounted
{'zone': 5, 'weight': 1952.0, 'ip': '<redacted>.204.20', 'id': 2311, 'meta': '
↪', 'device': 'disk5', 'port': 6201}
Mar 14 17:28:19 <redacted> container-replicator ERROR Remote drive not mounted
{'zone': 5, 'weight': 1952.0, 'ip': '<redacted>.204.20', 'id': 2311, 'meta': '
↪', 'device': 'disk5', 'port': 6201}
Mar 14 17:28:20 <redacted> container-replicator ERROR Remote drive not mounted
{'zone': 5, 'weight': 1952.0, 'ip': '<redacted>.204.119', 'id': 5481, 'meta':
↪'', 'device': 'disk6', 'port': 6201}
Mar 14 17:28:21 <redacted> container-replicator ERROR Remote drive not mounted
{'zone': 5, 'weight': 1952.0, 'ip': '<redacted>.204.20', 'id': 2311, 'meta': '
↪', 'device': 'disk5', 'port': 6201}
Mar 14 17:28:21 <redacted> container-replicator ERROR Remote drive not mounted
{'zone': 5, 'weight': 1952.0, 'ip': '<redacted>.204.20', 'id': 2311, 'meta': '
↪', 'device': 'disk5', 'port': 6201}
Mar 14 17:28:22 <redacted> container-replicator ERROR Remote drive not mounted
{'zone': 5, 'weight': 1952.0, 'ip': '<redacted>.204.20', 'id': 2311, 'meta': '
↪', 'device': 'disk5', 'port': 6201}
```

That is why this node has a lot of async pendings: a bunch of disks that are not mounted on <redacted> and <redacted>. There may be other issues, but clearing this up will likely drop the async pendings a fair bit, as other nodes will be having the same problem.

Assessing the availability risk when multiple storage servers are down

---

Note: This procedure will tell you if you have a problem, however, in practice you will find that you will not use this procedure frequently.

If three storage nodes (or, more precisely, three disks on three different storage nodes) are down, there is a small but nonzero probability that user objects, containers, or accounts will not be available.

Procedure

Note: swift has three rings: one each for objects, containers and accounts. This procedure should be run three times, each time specifying the appropriate `*.builder` file.

1. Determine whether all three nodes are in different Swift zones by running the ring builder on a proxy node to determine which zones the storage nodes are in. For example:

```
% sudo swift-ring-builder /etc/swift/object.builder
/etc/swift/object.builder, build version 1467
2097152 partitions, 3 replicas, 5 zones, 1320 devices, 0.02 balance
The minimum number of hours before a partition can be reassigned is 24
Devices:    id  zone     ip address    port     name  weight  partitions␣
↪balance meta
             0     1     <redacted>.4  6200     disk0 1708.00       4259 ␣
↪ -0.00
             1     1     <redacted>.4  6200     disk1 1708.00       4260 ␣
↪  0.02
             2     1     <redacted>.4  6200     disk2 1952.00       4868 ␣
↪  0.01
             3     1     <redacted>.4  6200     disk3 1952.00       4868 ␣
↪  0.01
             4     1     <redacted>.4  6200     disk4 1952.00       4867 ␣
↪ -0.01
```

2. Here, node <redacted>.4 is in zone 1. If two or more of the three nodes under consideration are in the same Swift zone, they do not have any ring partitions in common; there is little/no data availability risk if all three nodes are down.

3. If the nodes are in three distinct Swift zones it is necessary to whether the nodes have ring partitions in common. Run `swift-ring` builder again, this time with the `list_parts` option and specify the nodes under consideration. For example:

```
% sudo swift-ring-builder /etc/swift/object.builder list_parts <redacted>.
↪8 <redacted>.15 <redacted>.72.2
Partition   Matches
91          2
729         2
3754        2
3769        2
3947        2
5818        2
7918        2
8733        2
9509        2
10233       2
```

4. The `list_parts` option to the ring builder indicates how many ring partitions the nodes have in common. If, as in this case, the first entry in the list has a Matches column of 2 or less, there is no data availability risk if all three nodes are down.

5. If the Matches column has entries equal to 3, there is some data availability risk if all three nodes are down. The risk is generally small, and is proportional to the number of entries that have a 3 in the Matches column. For example:

```
Partition   Matches
26865         3
362367        3
745940        3
778715        3
797559        3
820295        3
822118        3
839603        3
852332        3
855965        3
858016        3
```

6. A quick way to count the number of rows with 3 matches is:

```
% sudo swift-ring-builder /etc/swift/object.builder list_parts <redacted>.
↪8 <redacted>.15 <redacted>.72.2 | grep "3$" | wc -l

30
```

7. In this case the nodes have 30 out of a total of 2097152 partitions in common; about 0.001%. In this case the risk is small/nonzero. Recall that a partition is simply a portion of the ring mapping space, not actual data. So having partitions in common is a necessary but not sufficient condition for data unavailability.

Note: We should not bring down a node for repair if it shows Matches entries of 3 with other nodes that are also down.

If three nodes that have 3 partitions in common are all down, there is a nonzero probability that data are unavailable and we should work to bring some or all of the nodes up ASAP.

Swift startup/shutdown

- Use reload - not stop/start/restart.
- Try to roll sets of servers (especially proxy) in groups of less than 20% of your servers.

5.7.4 Troubleshooting tips

Diagnose: Customer complains they receive a HTTP status 500 when trying to browse containers

This entry is prompted by a real customer issue and exclusively focused on how that problem was identified. There are many reasons why a http status of 500 could be returned. If there are no obvious problems with the swift object store, then it may be necessary to take a closer look at the users transactions. After finding the users swift account, you can search the swift proxy logs on each swift proxy server for transactions from this user. The linux `bzgrep` command can be used to search all the proxy log files on a node including the `.bz2` compressed files. For example:

```
$ PDSH_SSH_ARGS_APPEND="-o StrictHostKeyChecking=no" pdsh -l <yourusername> -
↪R ssh \
  -w <redacted>.68.[4-11,132-139 4-11,132-139],<redacted>.132.[4-11,132-139] \
  'sudo bzgrep -w AUTH_redacted-4962-4692-98fb-52ddda82a5af /var/log/swift/
↪proxy.log*' |  dshbak -c
.
.
----------------
<redacted>.132.6
----------------
Feb 29 08:51:57 sw-aw2az2-proxy011 proxy-server <redacted>.16.132
<redacted>.66.8 29/Feb/2012/08/51/57 GET /v1.0/AUTH_redacted-4962-4692-98fb-
↪52ddda82a5af
/%3Fformat%3Djson HTTP/1.0 404 - - <REDACTED>_4f4d50c5e4b064d88bd7ab82 - - -
tx429fc3be354f434ab7f9c6c4206c1dc3 - 0.0130
```

This shows a `GET` operation on the users account.

Note: The HTTP status returned is 404, Not found, rather than 500 as reported by the user.

Using the transaction ID, `tx429fc3be354f434ab7f9c6c4206c1dc3` you can search the swift object servers log files for this transaction ID:

```
$ PDSH_SSH_ARGS_APPEND="-o StrictHostKeyChecking=no" pdsh -l <yourusername> -
↪R ssh \
  -w <redacted>.72.[4-67|4-67],<redacted>.[4-67|4-67],<redacted>.[4-67|4-67],
↪<redacted>.204.[4-131] \
  'sudo bzgrep tx429fc3be354f434ab7f9c6c4206c1dc3 /var/log/swift/server.log*'␣
↪| dshbak -c
.
.
----------------
<redacted>.72.16
----------------
Feb 29 08:51:57 sw-aw2az1-object013 account-server <redacted>.132.6 - -

[29/Feb/2012:08:51:57 +0000|] "GET /disk9/198875/AUTH_redacted-4962-4692-98fb-
↪52ddda82a5af"
```

(continues on next page)

(continued from previous page)

```
404 - "tx429fc3be354f434ab7f9c6c4206c1dc3" "-" "-"

0.0016 ""
---------------
<redacted>.31
---------------
Feb 29 08:51:57 node-az2-object060 account-server <redacted>.132.6 - -
[29/Feb/2012:08:51:57 +0000|] "GET /disk6/198875/AUTH_redacted-4962-
4692-98fb-52ddda82a5af" 404 - "tx429fc3be354f434ab7f9c6c4206c1dc3" "-" "-" 0.
↪0011 ""
---------------
<redacted>.204.70
---------------

Feb 29 08:51:57 sw-aw2az3-object0067 account-server <redacted>.132.6 - -
[29/Feb/2012:08:51:57 +0000|] "GET /disk6/198875/AUTH_redacted-4962-
4692-98fb-52ddda82a5af" 404 - "tx429fc3be354f434ab7f9c6c4206c1dc3" "-" "-" 0.
↪0014 ""
```

Note: The 3 GET operations to 3 different object servers that hold the 3 replicas of this users account. Each `GET` returns a HTTP status of 404, Not found.

Next, use the `swift-get-nodes` command to determine exactly where the users account data is stored:

```
$ sudo swift-get-nodes /etc/swift/account.ring.gz AUTH_redacted-4962-4692-
↪98fb-52ddda82a5af
Account AUTH_redacted-4962-4692-98fb-52ddda82a5af
Container None
Object None

Partition 198875
Hash 1846d99185f8a0edaf65cfbf37439696

Server:Port Device <redacted>.31:6202 disk6
Server:Port Device <redacted>.204.70:6202 disk6
Server:Port Device <redacted>.72.16:6202 disk9
Server:Port Device <redacted>.204.64:6202 disk11 [Handoff]
Server:Port Device <redacted>.26:6202 disk11 [Handoff]
Server:Port Device <redacted>.72.27:6202 disk11 [Handoff]

curl -I -XHEAD "`http://<redacted>.31:6202/disk6/198875/AUTH_redacted-4962-
↪4692-98fb-52ddda82a5af"
<http://15.185.138.31:6202/disk6/198875/AUTH_db0050ad-4962-4692-98fb-
↪52ddda82a5af>`_
curl -I -XHEAD "`http://<redacted>.204.70:6202/disk6/198875/AUTH_redacted-
↪4962-4692-98fb-52ddda82a5af"
<http://15.185.204.70:6202/disk6/198875/AUTH_db0050ad-4962-4692-98fb-
↪52ddda82a5af>`_
```

(continues on next page)

(continued from previous page)

```
curl -I -XHEAD "`http://<redacted>.72.16:6202/disk9/198875/AUTH_redacted-4962-
↪4692-98fb-52ddda82a5af"
<http://15.185.72.16:6202/disk9/198875/AUTH_db0050ad-4962-4692-98fb-
↪52ddda82a5af>`_
curl -I -XHEAD "`http://<redacted>.204.64:6202/disk11/198875/AUTH_redacted-
↪4962-4692-98fb-52ddda82a5af"
<http://15.185.204.64:6202/disk11/198875/AUTH_db0050ad-4962-4692-98fb-
↪52ddda82a5af>`_ # [Handoff]
curl -I -XHEAD "`http://<redacted>.26:6202/disk11/198875/AUTH_redacted-4962-
↪4692-98fb-52ddda82a5af"
<http://15.185.136.26:6202/disk11/198875/AUTH_db0050ad-4962-4692-98fb-
↪52ddda82a5af>`_ # [Handoff]
curl -I -XHEAD "`http://<redacted>.72.27:6202/disk11/198875/AUTH_redacted-
↪4962-4692-98fb-52ddda82a5af"
<http://15.185.72.27:6202/disk11/198875/AUTH_db0050ad-4962-4692-98fb-
↪52ddda82a5af>`_ # [Handoff]

ssh <redacted>.31 "ls -lah /srv/node/disk6/accounts/198875/696/
↪1846d99185f8a0edaf65cfbf37439696/"
ssh <redacted>.204.70 "ls -lah /srv/node/disk6/accounts/198875/696/
↪1846d99185f8a0edaf65cfbf37439696/"
ssh <redacted>.72.16 "ls -lah /srv/node/disk9/accounts/198875/696/
↪1846d99185f8a0edaf65cfbf37439696/"
ssh <redacted>.204.64 "ls -lah /srv/node/disk11/accounts/198875/696/
↪1846d99185f8a0edaf65cfbf37439696/" # [Handoff]
ssh <redacted>.26 "ls -lah /srv/node/disk11/accounts/198875/696/
↪1846d99185f8a0edaf65cfbf37439696/" # [Handoff]
ssh <redacted>.72.27 "ls -lah /srv/node/disk11/accounts/198875/696/
↪1846d99185f8a0edaf65cfbf37439696/" # [Handoff]
```

Check each of the primary servers, <redacted>.31, <redacted>.204.70 and <redacted>.72.16, for this users account. For example on <redacted>.72.16:

```
$ ls -lah /srv/node/disk9/accounts/198875/696/
↪1846d99185f8a0edaf65cfbf37439696/
total 1.0M
drwxrwxrwx 2 swift swift 98 2012-02-23 14:49 .
drwxrwxrwx 3 swift swift 45 2012-02-03 23:28 ..
-rw------- 1 swift swift 15K 2012-02-23 14:49␣
↪1846d99185f8a0edaf65cfbf37439696.db
-rw-rw-rw- 1 swift swift 0 2012-02-23 14:49 1846d99185f8a0edaf65cfbf37439696.
↪db.pending
```

So this users account db, an sqlite db is present. Use sqlite to checkout the account:

```
$ sudo cp /srv/node/disk9/accounts/198875/696/
↪1846d99185f8a0edaf65cfbf37439696/1846d99185f8a0edaf65cfbf37439696.db /tmp
$ sudo sqlite3 /tmp/1846d99185f8a0edaf65cfbf37439696.db
sqlite> .mode line
sqlite> select * from account_stat;
```

(continues on next page)

(continued from previous page)

```
account = AUTH_redacted-4962-4692-98fb-52ddda82a5af
created_at = 1328311738.42190
put_timestamp = 1330000873.61411
delete_timestamp = 1330001026.00514
container_count = 0
object_count = 0
bytes_used = 0
hash = eb7e5d0ea3544d9def940b19114e8b43
id = 2de8c8a8-cef9-4a94-a421-2f845802fe90
status = DELETED
status_changed_at = 1330001026.00514
metadata =
```

Next try and find the DELETE operation for this account in the proxy server logs:

```
$ PDSH_SSH_ARGS_APPEND="-o StrictHostKeyChecking=no" pdsh -l <yourusername> -
↪R ssh \
  -w <redacted>.68.[4-11,132-139 4-11,132-139],<redacted>.132.[4-11,132-139|4-
↪11,132-139] \
  'sudo bzgrep AUTH_redacted-4962-4692-98fb-52ddda82a5af /var/log/swift/proxy.
↪log* \
  | grep -w DELETE | awk "{print $3,$10,$12}"' |- dshbak -c
.
.
Feb 23 12:43:46 sw-aw2az2-proxy001 proxy-server <redacted> <redacted>.66.7 23/
↪Feb/2012/12/43/46 DELETE /v1.0/AUTH_redacted-4962-4692-98fb-
52ddda82a5af/ HTTP/1.0 204 - Apache-HttpClient/4.1.2%20%28java%201.5%29
↪<REDACTED>_4f458ee4e4b02a869c3aad02 - - -
tx4471188b0b87406899973d297c55ab53 - 0.0086
```

From this you can see the operation that resulted in the account being deleted.

Procedure: Deleting objects

Simple case - deleting small number of objects and containers

---

Note: `swift-direct` is specific to the Hewlett Packard Enterprise Helion Public Cloud. Use `swiftly` as an alternative.

Note: Object and container names are in UTF8. Swift direct accepts UTF8 directly, not URL-encoded UTF8 (the REST API expects UTF8 and then URL-encoded). In practice cut and paste of foreign language strings to a terminal window will produce the right result.

Hint: Use the `head` command before any destructive commands.

To delete a small number of objects, log into any proxy node and proceed as follows:

Examine the object in question:

```
$ sudo -u swift /opt/hp/swift/bin/swift-direct head 132345678912345 container_
↪name obj_name
```

See if `X-Object-Manifest` or `X-Static-Large-Object` is set, then this is the manifest object and segment objects may be in another container.

If the `X-Object-Manifest` attribute is set, you need to find the name of the objects this means it is a DLO. For example, if `X-Object-Manifest` is `container2/seg-blah`, list the contents of the container container2 as follows:

```
$ sudo -u swift /opt/hp/swift/bin/swift-direct show 132345678912345 container2
```

Pick out the objects whose names start with `seg-blah`. Delete the segment objects as follows:

```
$ sudo -u swift /opt/hp/swift/bin/swift-direct delete 132345678912345␣
↪container2 seg-blah01
$ sudo -u swift /opt/hp/swift/bin/swift-direct delete 132345678912345␣
↪container2 seg-blah02
etc
```

If `X-Static-Large-Object` is set, you need to read the contents. Do this by:

- Using swift-get-nodes to get the details of the objects location.
- Change the `-X HEAD` to `-X GET` and run `curl` against one copy.
- This lists a JSON body listing containers and object names
- Delete the objects as described above for DLO segments

Once the segments are deleted, you can delete the object using `swift-direct` as described above.

Finally, use `swift-direct` to delete the container.

Procedure: Decommissioning swift nodes

Should Swift nodes need to be decommissioned (e.g.,, where they are being re-purposed), it is very important to follow the following steps.

1. In the case of object servers, follow the procedure for removing the node from the rings.
2. In the case of swift proxy servers, have the network team remove the node from the load balancers.
3. Open a network ticket to have the node removed from network firewalls.
4. Make sure that you remove the `/etc/swift` directory and everything in it.

5.8 OpenStack Swift Administrator Guide

5.8.1 Introduction to Object Storage

OpenStack Object Storage (swift) is used for redundant, scalable data storage using clusters of standardized servers to store petabytes of accessible data. It is a long-term storage system for large amounts of static data which can be retrieved and updated. Object Storage uses a distributed architecture with no central point of control, providing greater scalability, redundancy, and permanence. Objects are written to multiple hardware devices, with the OpenStack software responsible for ensuring data replication and integrity across the cluster. Storage clusters scale horizontally by adding new nodes. Should a node fail, OpenStack works to replicate its content from other active nodes. Because OpenStack uses software logic to ensure data replication and distribution across different devices, inexpensive commodity hard drives and servers can be used in lieu of more expensive equipment.

Object Storage is ideal for cost effective, scale-out storage. It provides a fully distributed, API-accessible storage platform that can be integrated directly into applications or used for backup, archiving, and data retention.

5.8.2 Features and benefits

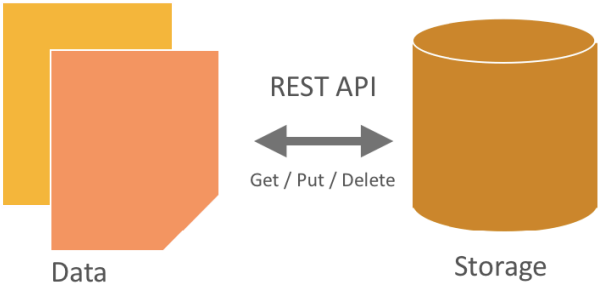
Features	Benefits
Leverages commodity hardware	No lock-in, lower price/GB.
HDD/node failure agnostic	Self-healing, reliable, data redundancy protects from failures.
Unlimited storage	Large and flat namespace, highly scalable read/write access, able to serve content directly from storage system.
Multi-dimensional scalability	Scale-out architecture: Scale vertically and horizontally-distributed storage. Backs up and archives large amounts of data with linear performance.
Account/container/object structure	No nesting, not a traditional file system: Optimized for scale, it scales to multiple petabytes and billions of objects.
Built-in replication 3× + data redundancy (compared with 2× on RAID)	A configurable number of accounts, containers and object copies for high availability.
Easily add capacity (unlike RAID resize)	Elastic data scaling with ease.
No central database	Higher performance, no bottlenecks.
RAID not required	Handle many small, random reads and writes efficiently.
Built-in management utilities	Account management: Create, add, verify, and delete users; Container management: Upload, download, and verify; Monitoring: Capacity, host, network, log trawling, and cluster health.
Drive auditing	Detect drive failures preempting data corruption.
Expiring objects	Users can set an expiration time or a TTL on an object to control access.
Direct object access	Enable direct browser access to content, such as for a control panel.
Realtime visibility into client requests	Know what users are requesting.
Supports S3 API	Utilize tools that were designed for the popular S3 API.
Restrict containers per account	Limit access to control usage by user.

5.8.3 Object Storage characteristics

The key characteristics of Object Storage are that:

- All objects stored in Object Storage have a URL.
- Storage Policies may be used to define different levels of durability for objects stored in the cluster. These policies support not only complete replicas but also erasure-coded fragments.
- All replicas or fragments for an object are stored in as-unique-as-possible zones to increase durability and availability.
- All objects have their own metadata.
- Developers interact with the object storage system through a RESTful HTTP API.
- Object data can be located anywhere in the cluster.
- The cluster scales by adding additional nodes without sacrificing performance, which allows a more cost-effective linear storage expansion than fork-lift upgrades.
- Data does not have to be migrated to an entirely new storage system.
- New nodes can be added to the cluster without downtime.
- Failed nodes and disks can be swapped out without downtime.
- It runs on industry-standard hardware, such as Dell, HP, and Supermicro.

Object Storage (swift)



Developers can either write directly to the Swift API or use one of the many client libraries that exist for all of the popular programming languages, such as Java, Python, Ruby, and C#. Amazon S3 and RackSpace Cloud Files users should be very familiar with Object Storage. Users new to object storage systems will have to adjust to a different approach and mindset than those required for a traditional filesystem.

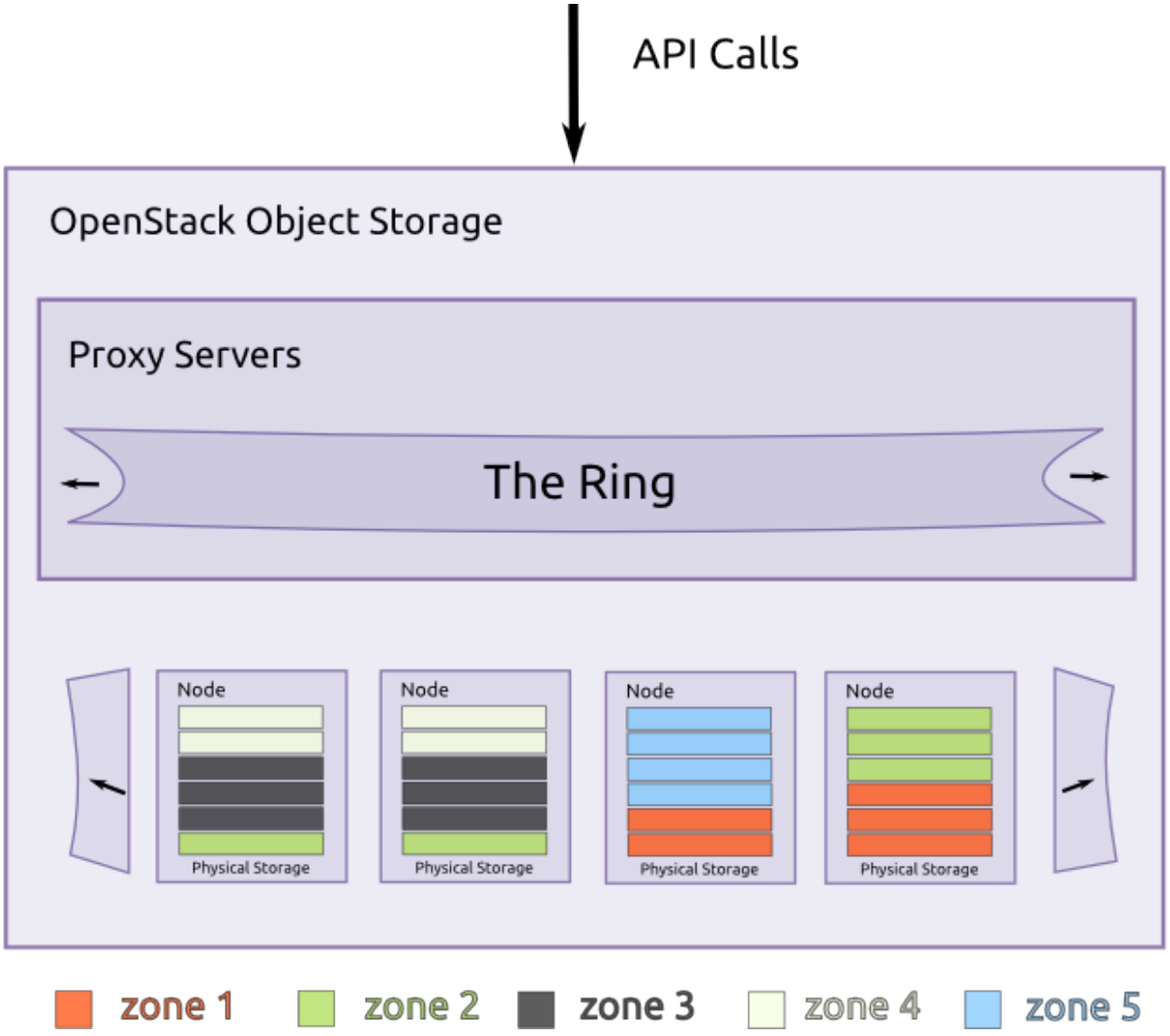
5.8.4 Components

Object Storage uses the following components to deliver high availability, high durability, and high concurrency:

- **Proxy servers** - Handle all of the incoming API requests.
- **Rings** - Map logical names of data to locations on particular disks.
- **Zones** - Isolate data from other zones. A failure in one zone does not impact the rest of the cluster as data replicates across zones.

- **Accounts and containers** - Each account and container are individual databases that are distributed across the cluster. An account database contains the list of containers in that account. A container database contains the list of objects in that container.
- **Objects** - The data itself.
- **Partitions** - A partition stores objects, account databases, and container databases and helps manage locations where data lives in the cluster.

Object Storage building blocks



Proxy servers

Proxy servers are the public face of Object Storage and handle all of the incoming API requests. Once a proxy server receives a request, it determines the storage node based on the objects URL, for example: `https://swift.example.com/v1/account/container/object`. Proxy servers also coordinate responses, handle failures, and coordinate timestamps.

Proxy servers use a shared-nothing architecture and can be scaled as needed based on projected workloads. A minimum of two proxy servers should be deployed behind a separately-managed load balancer. If one proxy server fails, the others take over.

Rings

A ring represents a mapping between the names of entities stored in the cluster and their physical locations on disks. There are separate rings for accounts, containers, and objects. When components of the system need to perform an operation on an object, container, or account, they need to interact with the corresponding ring to determine the appropriate location in the cluster.

The ring maintains this mapping using zones, devices, partitions, and replicas. Each partition in the ring is replicated, by default, three times across the cluster, and partition locations are stored in the mapping maintained by the ring. The ring is also responsible for determining which devices are used as handoffs in failure scenarios.

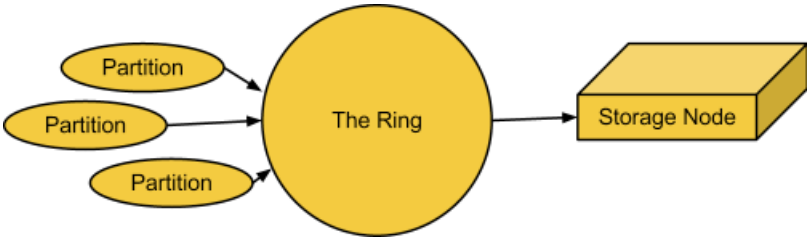
Data can be isolated into zones in the ring. Each partition replica will try to reside in a different zone. A zone could represent a drive, a server, a cabinet, a switch, or even a data center.

The partitions of the ring are distributed among all of the devices in the Object Storage installation. When partitions need to be moved around (for example, if a device is added to the cluster), the ring ensures that a minimum number of partitions are moved at a time, and only one replica of a partition is moved at a time.

You can use weights to balance the distribution of partitions on drives across the cluster. This can be useful, for example, when differently sized drives are used in a cluster.

The ring is used by the proxy server and several background processes (like replication).

The ring



These rings are externally managed. The server processes themselves do not modify the rings, they are instead given new rings modified by other tools.

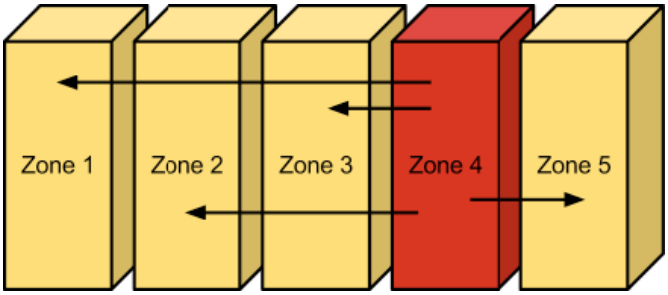
The ring uses a configurable number of bits from an `MD5` hash for a path as a partition index that designates a device. The number of bits kept from the hash is known as the partition power, and 2 to the partition power indicates the partition count. Partitioning the full `MD5` hash ring allows other parts of the cluster to work in batches of items at once which ends up either more efficient or at least less complex than working with each item separately or the entire cluster all at once.

Another configurable value is the replica count, which indicates how many of the partition-device assignments make up a single ring. For a given partition index, each replicas device will not be in the same zone as any other replicas device. Zones can be used to group devices based on physical locations, power separations, network separations, or any other attribute that would improve the availability of multiple replicas at the same time.

Zones

Object Storage allows configuring zones in order to isolate failure boundaries. If possible, each data replica resides in a separate zone. At the smallest level, a zone could be a single drive or a grouping of a few drives. If there were five object storage servers, then each server would represent its own zone. Larger deployments would have an entire rack (or multiple racks) of object servers, each representing a zone. The goal of zones is to allow the cluster to tolerate significant outages of storage servers without losing all replicas of the data.

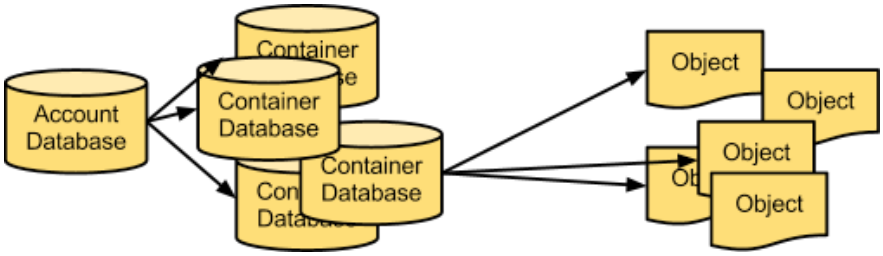
Zones



Accounts and containers

Each account and container is an individual SQLite database that is distributed across the cluster. An account database contains the list of containers in that account. A container database contains the list of objects in that container.

Accounts and containers



To keep track of object data locations, each account in the system has a database that references all of its containers, and each container database references each object.

Partitions

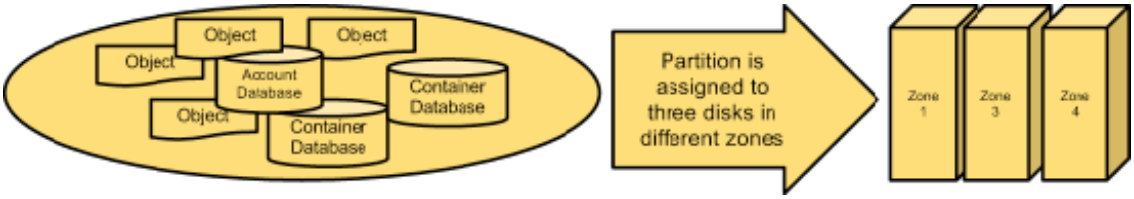
A partition is a collection of stored data. This includes account databases, container databases, and objects. Partitions are core to the replication system.

Think of a partition as a bin moving throughout a fulfillment center warehouse. Individual orders get thrown into the bin. The system treats that bin as a cohesive entity as it moves throughout the system. A bin is easier to deal with than many little things. It makes for fewer moving parts throughout the system.

System replicators and object uploads/downloads operate on partitions. As the system scales up, its behavior continues to be predictable because the number of partitions is a fixed number.

Implementing a partition is conceptually simple: a partition is just a directory sitting on a disk with a corresponding hash table of what it contains.

Partitions



Replicators

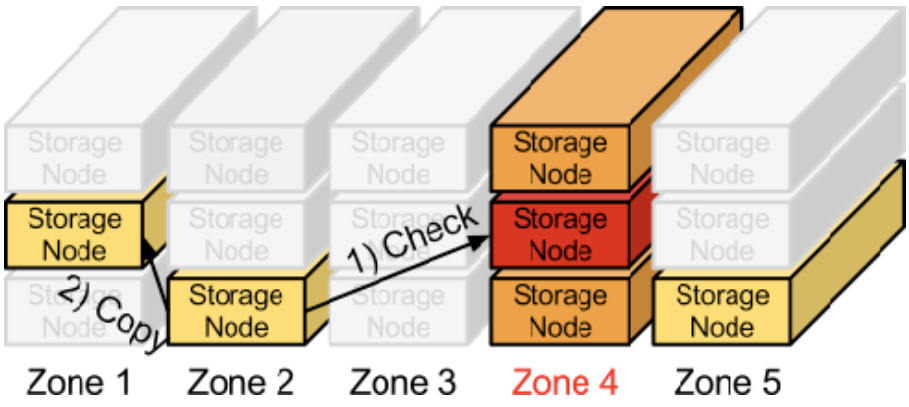
In order to ensure that there are three copies of the data everywhere, replicators continuously examine each partition. For each local partition, the replicator compares it against the replicated copies in the other zones to see if there are any differences.

The replicator knows if replication needs to take place by examining hashes. A hash file is created for each partition, which contains hashes of each directory in the partition. For a given partition, the hash files for each of the partitions copies are compared. If the hashes are different, then it is time to replicate, and the directory that needs to be replicated is copied over.

This is where partitions come in handy. With fewer things in the system, larger chunks of data are transferred around (rather than lots of little TCP connections, which is inefficient) and there is a consistent number of hashes to compare.

The cluster has an eventually-consistent behavior where old data may be served from partitions that missed updates, but replication will cause all partitions to converge toward the newest data.

Replication



If a zone goes down, one of the nodes containing a replica notices and proactively copies data to a handoff location.

Use cases

The following sections show use cases for object uploads and downloads and introduce the components.

Upload

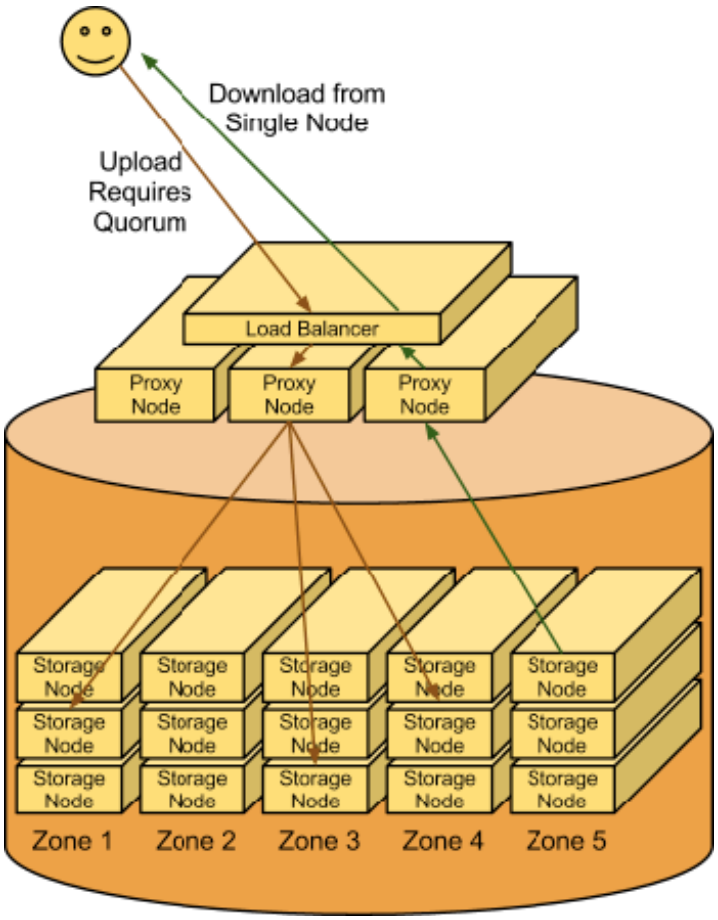
A client uses the REST API to make a HTTP request to PUT an object into an existing container. The cluster receives the request. First, the system must figure out where the data is going to go. To do this, the account name, container name, and object name are all used to determine the partition where this object should live.

Then a lookup in the ring figures out which storage nodes contain the partitions in question.

The data is then sent to each storage node where it is placed in the appropriate partition. At least two of the three writes must be successful before the client is notified that the upload was successful.

Next, the container database is updated asynchronously to reflect that there is a new object in it.

Object Storage in use



Download

A request comes in for an account/container/object. Using the same consistent hashing, the partition index is determined. A lookup in the ring reveals which storage nodes contain that partition. A request is made to one of the storage nodes to fetch the object and, if that fails, requests are made to the other nodes.

5.8.5 Ring-builder

Use the swift-ring-builder utility to build and manage rings. This utility assigns partitions to devices and writes an optimized Python structure to a gzipped, serialized file on disk for transmission to the servers. The server processes occasionally check the modification time of the file and reload in-memory copies of the ring structure as needed. If you use a slightly older version of the ring, one of the three replicas for a partition subset will be incorrect because of the way the ring-builder manages changes to the ring. You can work around this issue.

The ring-builder also keeps its own builder file with the ring information and additional data required to build future rings. It is very important to keep multiple backup copies of these builder files. One option is to copy the builder files out to every server while copying the ring files themselves. Another is to upload the builder files into the cluster itself. If you lose the builder file, you have to create a new ring from scratch. Nearly all partitions would be assigned to different devices and, therefore, nearly all of the stored data would have to be replicated to new locations. So, recovery from a builder file loss is possible, but data would be unreachable for an extended time.

Ring data structure

The ring data structure consists of three top level fields: a list of devices in the cluster, a list of lists of device ids indicating partition to device assignments, and an integer indicating the number of bits to shift an MD5 hash to calculate the partition for the hash.

Partition assignment list

This is a list of `array('H')` of devices ids. The outermost list contains an `array('H')` for each replica. Each `array('H')` has a length equal to the partition count for the ring. Each integer in the `array('H')` is an index into the above list of devices. The partition list is known internally to the Ring class as `_replica2part2dev_id`.

So, to create a list of device dictionaries assigned to a partition, the Python code would look like:

```
devices = [self.devs[part2dev_id[partition]] for
part2dev_id in self._replica2part2dev_id]
```

That code is a little simplistic because it does not account for the removal of duplicate devices. If a ring has more replicas than devices, a partition will have more than one replica on a device.

`array('H')` is used for memory conservation as there may be millions of partitions.

Overload

The ring builder tries to keep replicas as far apart as possible while still respecting device weights. When it can not do both, the overload factor determines what happens. Each device takes an extra fraction of its desired partitions to allow for replica dispersion; after that extra fraction is exhausted, replicas are placed closer together than optimal.

The overload factor lets the operator trade off replica dispersion (durability) against data dispersion (uniform disk usage).

The default overload factor is 0, so device weights are strictly followed.

With an overload factor of 0.1, each device accepts 10% more partitions than it otherwise would, but only if it needs to maintain partition dispersion.

For example, consider a 3-node cluster of machines with equal-size disks; node A has 12 disks, node B has 12 disks, and node C has 11 disks. The ring has an overload factor of 0.1 (10%).

Without the overload, some partitions would end up with replicas only on nodes A and B. However, with the overload, every device can accept up to 10% more partitions for the sake of dispersion. The missing disk in C means there is one disks worth of partitions to spread across the remaining 11 disks, which gives each disk in C an extra 9.09% load. Since this is less than the 10% overload, there is one replica of each partition on each node.

However, this does mean that the disks in node C have more data than the disks in nodes A and B. If 80% full is the warning threshold for the cluster, node Cs disks reach 80% full while A and Bs disks are only 72.7% full.

Replica counts

To support the gradual change in replica counts, a ring can have a real number of replicas and is not restricted to an integer number of replicas.

A fractional replica count is for the whole ring and not for individual partitions. It indicates the average number of replicas for each partition. For example, a replica count of 3.2 means that 20 percent of partitions have four replicas and 80 percent have three replicas.

The replica count is adjustable. For example:

```
$ swift-ring-builder account.builder set_replicas 4
$ swift-ring-builder account.builder rebalance
```

You must rebalance the replica ring in globally distributed clusters. Operators of these clusters generally want an equal number of replicas and regions. Therefore, when an operator adds or removes a region, the operator adds or removes a replica. Removing unneeded replicas saves on the cost of disks.

You can gradually increase the replica count at a rate that does not adversely affect cluster performance. For example:

```
$ swift-ring-builder object.builder set_replicas 3.01
$ swift-ring-builder object.builder rebalance
<distribute rings and wait>...

$ swift-ring-builder object.builder set_replicas 3.02
$ swift-ring-builder object.builder rebalance
<distribute rings and wait>...
```

Changes take effect after the ring is rebalanced. Therefore, if you intend to change from 3 replicas to 3.01 but you accidentally type 2.01, no data is lost.

Additionally, the **`swift-ring-builder X.builder create`** command can now take a decimal argument for the number of replicas.

Partition shift value

The partition shift value is known internally to the Ring class as `_part_shift`. This value is used to shift an MD5 hash to calculate the partition where the data for that hash should reside. Only the top four bytes of the hash is used in this process. For example, to compute the partition for the `/account/container/object` path using Python:

```
partition = unpack_from('>I',
md5('/account/container/object').digest())[0] >>
self._part_shift
```

For a ring generated with part_power P, the partition shift value is `32 - P`.

Build the ring

The ring builder process includes these high-level steps:

1. The utility calculates the number of partitions to assign to each device based on the weight of the device. For example, for a partition at the power of 20, the ring has 1,048,576 partitions. One thousand devices of equal weight each want 1,048.576 partitions. The devices are sorted by the number of partitions they desire and kept in order throughout the initialization process.

   Note: Each device is also assigned a random tiebreaker value that is used when two devices desire the same number of partitions. This tiebreaker is not stored on disk anywhere, and so two different rings created with the same parameters will have different partition assignments. For repeatable partition assignments, `RingBuilder.rebalance()` takes an optional seed value that seeds the Python pseudo-random number generator.

   ---

2. The ring builder assigns each partition replica to the device that requires most partitions at that point while keeping it as far away as possible from other replicas. The ring builder prefers to assign a replica to a device in a region that does not already have a replica. If no such region is available, the ring builder searches for a device in a different zone, or on a different server. If it does not find one, it looks for a device with no replicas. Finally, if all options are exhausted, the ring builder assigns the replica to the device that has the fewest replicas already assigned.

   Note: The ring builder assigns multiple replicas to one device only if the ring has fewer devices than it has replicas.

   ---

3. When building a new ring from an old ring, the ring builder recalculates the desired number of partitions that each device wants.
4. The ring builder unassigns partitions and gathers these partitions for reassignment, as follows:
   - The ring builder unassigns any assigned partitions from any removed devices and adds these partitions to the gathered list.
   - The ring builder unassigns any partition replicas that can be spread out for better durability and adds these partitions to the gathered list.
   - The ring builder unassigns random partitions from any devices that have more partitions than they need and adds these partitions to the gathered list.

5. The ring builder reassigns the gathered partitions to devices by using a similar method to the one described previously.
6. When the ring builder reassigns a replica to a partition, the ring builder records the time of the reassignment. The ring builder uses this value when it gathers partitions for reassignment so that no partition is moved twice in a configurable amount of time. The RingBuilder class knows this configurable amount of time as `min_part_hours`. The ring builder ignores this restriction for replicas of partitions on removed devices because removal of a device happens on device failure only, and reassignment is the only choice.

These steps do not always perfectly rebalance a ring due to the random nature of gathering partitions for reassignment. To help reach a more balanced ring, the rebalance process is repeated until near perfect (less than 1 percent off) or when the balance does not improve by at least 1 percent (indicating we probably cannot get perfect balance due to wildly imbalanced zones or too many partitions recently moved).

5.8.6 Cluster architecture

Access tier

Large-scale deployments segment off an access tier, which is considered the Object Storage systems central hub. The access tier fields the incoming API requests from clients and moves data in and out of the system. This tier consists of front-end load balancers, ssl-terminators, and authentication services. It runs the (distributed) brain of the Object Storage system: the proxy server processes.

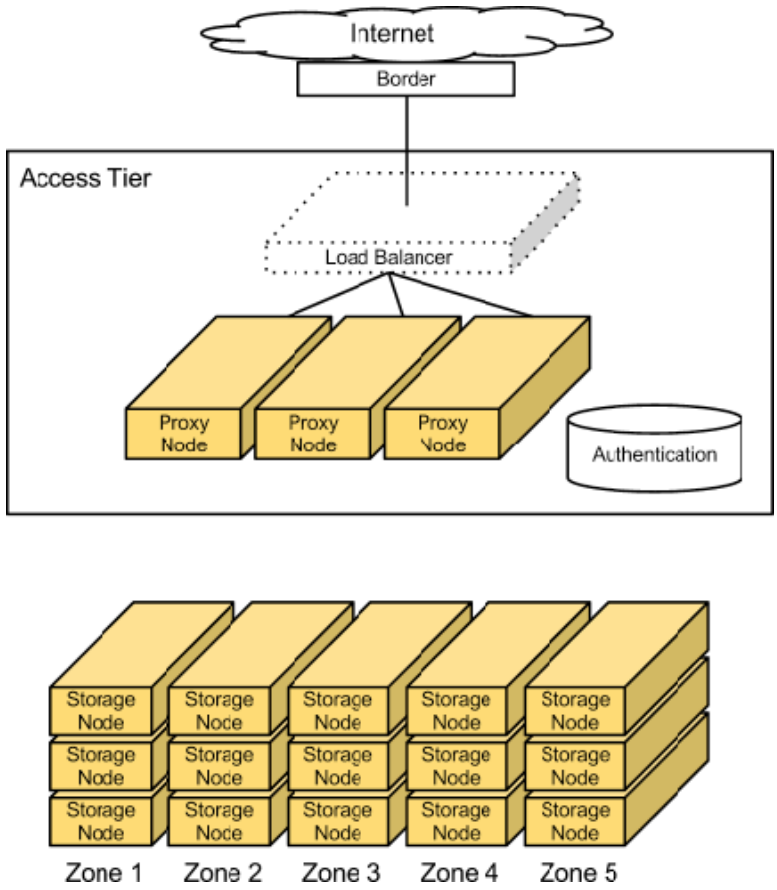
Note: If you want to use OpenStack Identity API v3 for authentication, you have the following options available in `/etc/swift/dispersion.conf`: `auth_version`, `user_domain_name`, `project_domain_name`, and `project_name`.

Object Storage architecture

Because access servers are collocated in their own tier, you can scale out read/write access regardless of the storage capacity. For example, if a cluster is on the public Internet, requires SSL termination, and has a high demand for data access, you can provision many access servers. However, if the cluster is on a private network and used primarily for archival purposes, you need fewer access servers.

Since this is an HTTP addressable storage service, you may incorporate a load balancer into the access tier.

Typically, the tier consists of a collection of 1U servers. These machines use a moderate amount of RAM and are network I/O intensive. Since these systems field each incoming API request, you should provision them with two high-throughput (10GbE) interfaces - one for the incoming front-end requests and the other for the back-end access to the object storage nodes to put and fetch data.



Factors to consider

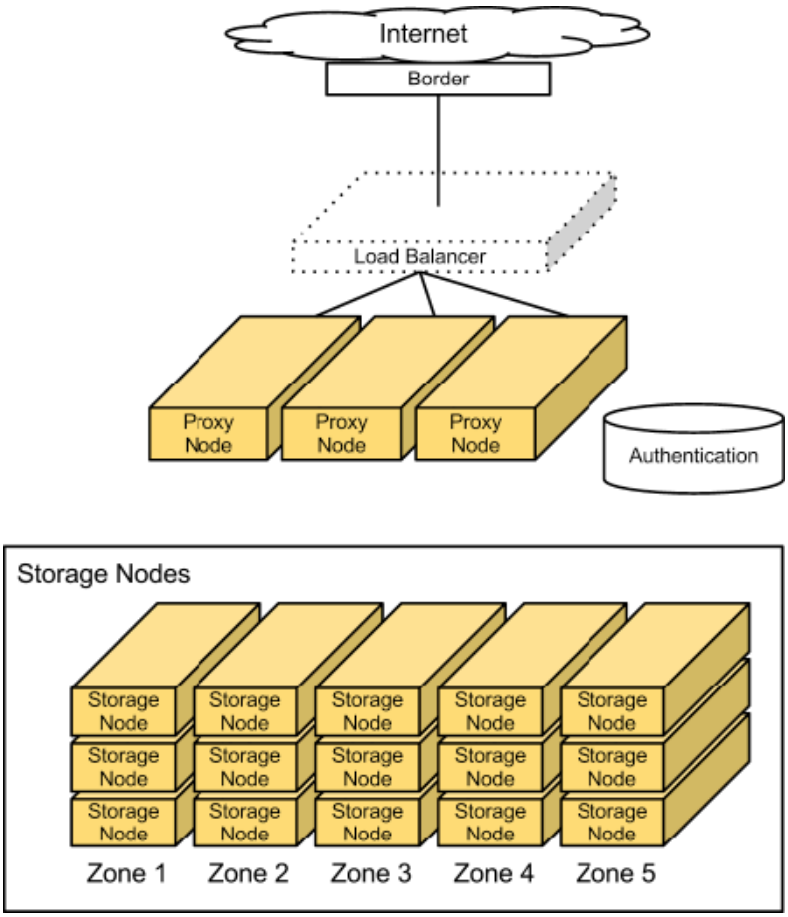
For most publicly facing deployments as well as private deployments available across a wide-reaching corporate network, you use SSL to encrypt traffic to the client. SSL adds significant processing load to establish sessions between clients, which is why you have to provision more capacity in the access layer. SSL may not be required for private deployments on trusted networks.

Storage nodes

In most configurations, each of the five zones should have an equal amount of storage capacity. Storage nodes use a reasonable amount of memory and CPU. Metadata needs to be readily available to return objects quickly. The object stores run services not only to field incoming requests from the access tier, but to also run replicators, auditors, and reapers. You can provision storage nodes with single gigabit or 10 gigabit network interface depending on the expected workload and desired performance, although it may be desirable to isolate replication traffic with a second interface.

Object Storage (swift)

Currently, a 2ăTB or 3ăTB SATA disk delivers good performance for the price. You can use desktop-grade drives if you have responsive remote hands in the datacenter and enterprise-grade drives if you dont.



Factors to consider

You should keep in mind the desired I/O performance for single-threaded requests. This system does not use RAID, so a single disk handles each request for an object. Disk performance impacts single-threaded response rates.

To achieve apparent higher throughput, the object storage system is designed to handle concurrent uploads/downloads. The network I/O capacity (1GbE, bonded 1GbE pair, or 10GbE) should match your desired concurrent throughput needs for reads and writes.

5.8.7 Replication

Because each replica in Object Storage functions independently and clients generally require only a simple majority of nodes to respond to consider an operation successful, transient failures like network partitions can quickly cause replicas to diverge. These differences are eventually reconciled by asynchronous, peer-to-peer replicator processes. The replicator processes traverse their local file systems and concurrently perform operations in a manner that balances load across physical disks.

Replication uses a push model, with records and files generally only being copied from local to remote replicas. This is important because data on the node might not belong there (as in the case of hand offs and ring changes), and a replicator cannot know which data it should pull in from elsewhere in the cluster. Any node that contains data must ensure that data gets to where it belongs. The ring handles replica placement.

To replicate deletions in addition to creations, every deleted record or file in the system is marked by a

tombstone. The replication process cleans up tombstones after a time period known as the `consistency window`. This window defines the duration of the replication and how long transient failure can remove a node from the cluster. Tombstone cleanup must be tied to replication to reach replica convergence.

If a replicator detects that a remote drive has failed, the replicator uses the `get_more_nodes` interface for the ring to choose an alternate node with which to synchronize. The replicator can maintain desired levels of replication during disk failures, though some replicas might not be in an immediately usable location.

Note: The replicator does not maintain desired levels of replication when failures such as entire node failures occur; most failures are transient.

The main replication types are:

- **Database replication** Replicates containers and objects.
- **Object replication** Replicates object data.

Database replication

Database replication completes a low-cost hash comparison to determine whether two replicas already match. Normally, this check can quickly verify that most databases in the system are already synchronized. If the hashes differ, the replicator synchronizes the databases by sharing records added since the last synchronization point.

This synchronization point is a high water mark that notes the last record at which two databases were known to be synchronized, and is stored in each database as a tuple of the remote database ID and record ID. Database IDs are unique across all replicas of the database, and record IDs are monotonically increasing integers. After all new records are pushed to the remote database, the entire synchronization table of the local database is pushed, so the remote database can guarantee that it is synchronized with everything with which the local database was previously synchronized.

If a replica is missing, the whole local database file is transmitted to the peer by using rsync(1) and is assigned a new unique ID.

In practice, database replication can process hundreds of databases per concurrency setting per second (up to the number of available CPUs or disks) and is bound by the number of database transactions that must be performed.

Object replication

The initial implementation of object replication performed an rsync to push data from a local partition to all remote servers where it was expected to reside. While this worked at small scale, replication times skyrocketed once directory structures could no longer be held in RAM. This scheme was modified to save a hash of the contents for each suffix directory to a per-partition hashes file. The hash for a suffix directory is no longer valid when the contents of that suffix directory is modified.

The object replication process reads in hash files and calculates any invalidated hashes. Then, it transmits the hashes to each remote server that should hold the partition, and only suffix directories with differing hashes on the remote server are rsynced. After pushing files to the remote server, the replication process notifies it to recalculate hashes for the rsynced suffix directories.

The number of uncached directories that object replication must traverse, usually as a result of invalidated suffix directory hashes, impedes performance. To provide acceptable replication speeds, object replication is designed to invalidate around 2 percent of the hash space on a normal node each day.

5.8.8 Large object support

Object Storage (swift) uses segmentation to support the upload of large objects. By default, Object Storage limits the download size of a single object to 5GB. Using segmentation, uploading a single object is virtually unlimited. The segmentation process works by fragmenting the object, and automatically creating a file that sends the segments together as a single object. This option offers greater upload speed with the possibility of parallel uploads.

Large objects

The large object is comprised of two types of objects:

- **Segment objects** store the object content. You can divide your content into segments, and upload each segment into its own segment object. Segment objects do not have any special features. You create, update, download, and delete segment objects just as you would normal objects.
- A **manifest object** links the segment objects into one logical large object. When you download a manifest object, Object Storage concatenates and returns the contents of the segment objects in the response body of the request. The manifest object types are:
  - **Static large objects**
  - **Dynamic large objects**

To find out more information on large object support, see *Large Object Support* in the developer documentation.

5.8.9 Object Auditor

On system failures, the XFS file system can sometimes truncate files it is trying to write and produce zero-byte files. The object-auditor will catch these problems but in the case of a system crash it is advisable to run an extra, less rate limited sweep, to check for these specific files. You can run this command as follows:

```
$ swift-object-auditor /path/to/object-server/config/file.conf once -z 1000
```

Note: -z means to only check for zero-byte files at 1000 files per second.

It is useful to run the object auditor on a specific device or set of devices. You can run the object-auditor once as follows:

```
$ swift-object-auditor /path/to/object-server/config/file.conf once \
  --devices=sda,sdb
```

Note: This will run the object auditor on only the `sda` and `sdb` devices. This parameter accepts a comma-separated list of values.

5.8.10 Erasure coding

Erasure coding is a set of algorithms that allows the reconstruction of missing data from a set of original data. In theory, erasure coding uses less capacity with similar durability characteristics as replicas. From an application perspective, erasure coding support is transparent. Object Storage (swift) implements erasure coding as a Storage Policy. See *Storage Policies* for more details.

There is no external API related to erasure coding. Create a container using a Storage Policy; the interaction with the cluster is the same as any other durability policy. Because support implements as a Storage Policy, you can isolate all storage devices that associate with your clusters erasure coding capability. It is entirely possible to share devices between storage policies, but for erasure coding it may make more sense to use not only separate devices but possibly even entire nodes dedicated for erasure coding.

5.8.11 Account reaper

The purpose of the account reaper is to remove data from the deleted accounts.

A reseller marks an account for deletion by issuing a `DELETE` request on the accounts storage URL. This action sets the `status` column of the account_stat table in the account database and replicas to `DELETED`, marking the accounts data for deletion.

Typically, a specific retention time or undelete are not provided. However, you can set a `delay_reaping` value in the `[account-reaper]` section of the `account-server.conf` file to delay the actual deletion of data. At this time, to undelete you have to update the account database replicas directly, set the status column to an empty string and update the put_timestamp to be greater than the delete_timestamp.

Note: It is on the development to-do list to write a utility that performs this task, preferably through a REST call.

The account reaper runs on each account server and scans the server occasionally for account databases marked for deletion. It only fires up on the accounts for which the server is the primary node, so that multiple account servers arent trying to do it simultaneously. Using multiple servers to delete one account might improve the deletion speed but requires coordination to avoid duplication. Speed really is not a big concern with data deletion, and large accounts arent deleted often.

Deleting an account is simple. For each account container, all objects are deleted and then the container is deleted. Deletion requests that fail will not stop the overall process but will cause the overall process to fail eventually (for example, if an object delete times out, you will not be able to delete the container or the account). The account reaper keeps trying to delete an account until it is empty, at which point the database reclaim process within the db_replicator will remove the database files.

A persistent error state may prevent the deletion of an object or container. If this happens, you will see a message in the log, for example:

```
Account <name> has not been reaped since <date>
```

You can control when this is logged with the `reap_warn_after` value in the `[account-reaper]` section of the `account-server.conf` file. The default value is 30 days.

5.8.12 Configure project-specific image locations with Object Storage

For some deployers, it is not ideal to store all images in one place to enable all projects and users to access them. You can configure the Image service to store image data in project-specific image locations. Then, only the following projects can use the Image service to access the created image:

- The project who owns the image
- Projects that are defined in `swift_store_admin_tenants` and that have admin-level accounts

To configure project-specific image locations

1. Configure swift as your `default_store` in the `glance-api.conf` file.
2. Set these configuration options in the `glance-api.conf` file:
   - **swift_store_multi_tenant** Set to `True` to enable tenant-specific storage locations. Default is `False`.
   - **swift_store_admin_tenants** Specify a list of tenant IDs that can grant read and write access to all Object Storage containers that are created by the Image service.

With this configuration, images are stored in an Object Storage service (swift) endpoint that is pulled from the service catalog for the authenticated user.

5.8.13 Object Storage monitoring

Note: This section was excerpted from a blog post by Darrell Bishop and has since been edited.

An OpenStack Object Storage cluster is a collection of many daemons that work together across many nodes. With so many different components, you must be able to tell what is going on inside the cluster. Tracking server-level meters like CPU utilization, load, memory consumption, disk usage and utilization, and so on is necessary, but not sufficient.

Swift Recon

The Swift Recon middleware (see *Cluster Telemetry and Monitoring*) provides general machine statistics, such as load average, socket statistics, `/proc/meminfo` contents, as well as Swift-specific meters:

- The `MD5` sum of each ring file.
- The most recent object replication time.
- Count of each type of quarantined file: Account, container, or object.
- Count of async_pendings (deferred container updates) on disk.

Swift Recon is middleware that is installed in the object servers pipeline and takes one required option: A local cache directory. To track `async_pendings`, you must set up an additional cron job for each object server. You access data by either sending HTTP requests directly to the object server or using the `swift-recon` command-line client.

There are Object Storage cluster statistics but the typical server meters overlap with existing server monitoring systems. To get the Swift-specific meters into a monitoring system, they must be polled. Swift Recon acts as a middleware meters collector. The process that feeds meters to your statistics system, such as `collectd` and `gmond`, should already run on the storage node. You can choose to either talk to Swift Recon or collect the meters directly.

Swift-Informant

Swift-Informant middleware (see swift-informant) has real-time visibility into Object Storage client requests. It sits in the pipeline for the proxy server, and after each request to the proxy server it sends three meters to a `StatsD` server:

- A counter increment for a meter like `obj.GET.200` or `cont.PUT.404`.
- Timing data for a meter like `acct.GET.200` or `obj.GET.200`. [The README says the meters look like `duration.acct.GET.200`, but I do not see the `duration` in the code. I am not sure what the Etsy server does but our StatsD server turns timing meters into five derivative meters with new segments appended, so it probably works as coded. The first meter turns into `acct.GET.200.lower`, `acct.GET.200.upper`, `acct.GET.200.mean`, `acct.GET.200.upper_90`, and `acct.GET.200.count`].
- A counter increase by the bytes transferred for a meter like `tfer.obj.PUT.201`.

This is used for receiving information on the quality of service clients experience with the timing meters, as well as sensing the volume of the various modifications of a request server type, command, and response code. Swift-Informant requires no change to core Object Storage code because it is implemented as middleware. However, it gives no insight into the workings of the cluster past the proxy server. If the responsiveness of one storage node degrades, you can only see that some of the requests are bad, either as high latency or error status codes.

Statsdlog

The Statsdlog project increments StatsD counters based on logged events. Like Swift-Informant, it is also non-intrusive, however statsdlog can track events from all Object Storage daemons, not just proxy-server. The daemon listens to a UDP stream of syslog messages, and StatsD counters are incremented when a log line matches a regular expression. Meter names are mapped to regex match patterns in a JSON file, allowing flexible configuration of what meters are extracted from the log stream.

Currently, only the first matching regex triggers a StatsD counter increment, and the counter is always incremented by one. There is no way to increment a counter by more than one or send timing data to StatsD based on the log line content. The tool could be extended to handle more meters for each line and data extraction, including timing data. But a coupling would still exist between the log textual format and the log parsing regexes, which would themselves be more complex to support multiple matches for each line and data extraction. Also, log processing introduces a delay between the triggering event and sending the data to StatsD. It would be preferable to increment error counters where they occur and send timing data as soon as it is known to avoid coupling between a log string and a parsing regex and prevent a time delay between events and sending data to StatsD.

The next section describes another method for gathering Object Storage operational meters.

Swift StatsD logging

StatsD (see Measure Anything, Measure Everything) was designed for application code to be deeply instrumented. Meters are sent in real-time by the code that just noticed or did something. The overhead of sending a meter is extremely low: a `sendto` of one UDP packet. If that overhead is still too high, the StatsD client library can send only a random portion of samples and StatsD approximates the actual number when flushing meters upstream.

To avoid the problems inherent with middleware-based monitoring and after-the-fact log processing, the sending of StatsD meters is integrated into Object Storage itself. Details of the meters tracked are in the *Administrators Guide*.

The sending of meters is integrated with the logging framework. To enable, configure `log_statsd_host` in the relevant config file. You can also specify the port and a default sample rate. The specified default sample rate is used unless a specific call to a statsd logging method (see the list below) overrides it. Currently, no logging calls override the sample rate, but it is conceivable that some meters may require accuracy (`sample_rate=1`) while others may not.

```
[DEFAULT]
# ...
log_statsd_host = 127.0.0.1
log_statsd_port = 8125
log_statsd_default_sample_rate = 1
```

Then the LogAdapter object returned by `get_logger()`, usually stored in `self.logger`, has these new methods:

- `update_stats(self, metric, amount, sample_rate=1)` Increments the supplied meter by the given amount. This is used when you need to add or subtract more that one from a counter, like incrementing `suffix.hashes` by the number of computed hashes in the object replicator.
- `increment(self, metric, sample_rate=1)` Increments the given counter meter by one.
- `decrement(self, metric, sample_rate=1)` Lowers the given counter meter by one.
- `timing(self, metric, timing_ms, sample_rate=1)` Record that the given meter took the supplied number of milliseconds.
- `timing_since(self, metric, orig_time, sample_rate=1)` Convenience method to record a timing meter whose value is now minus an existing timestamp.

Note: These logging methods may safely be called anywhere you have a logger object. If StatsD logging has not been configured, the methods are no-ops. This avoids messy conditional logic each place a meter is recorded. These example usages show the new logging methods:

```
# swift/obj/replicator.py
def update(self, job):
     # ...
    begin = time.time()
    try:
        hashed, local_hash = tpool.execute(tpooled_get_hashes, job['path'],
                do_listdir=(self.replication_count % 10) == 0,
                reclaim_age=self.reclaim_age)
        # See tpooled_get_hashes "Hack".
```

(continues on next page)

(continued from previous page)

```
        if isinstance(hashed, BaseException):
            raise hashed
        self.suffix_hash += hashed
        self.logger.update_stats('suffix.hashes', hashed)
        # ...
    finally:
        self.partition_times.append(time.time() - begin)
        self.logger.timing_since('partition.update.timing', begin)
```

```
# swift/container/updater.py
def process_container(self, dbfile):
    # ...
    start_time = time.time()
    # ...
        for event in events:
            if 200 <= event.wait() < 300:
                successes += 1
            else:
                failures += 1
        if successes > failures:
          self.logger.increment('successes')
            # ...
        else:
            self.logger.increment('failures')
            # ...
        # Only track timing data for attempted updates:
        self.logger.timing_since('timing', start_time)
    else:
        self.logger.increment('no_changes')
        self.no_changes += 1
```

5.8.14 Troubleshoot Object Storage

For Object Storage, everything is logged in `/var/log/syslog` (or `messages` on some distros). Several settings enable further customization of logging, such as `log_name`, `log_facility`, and `log_level`, within the object server configuration files.

Drive failure

Problem

Drive failure can prevent Object Storage performing replication.

Solution

In the event that a drive has failed, the first step is to make sure the drive is unmounted. This will make it easier for Object Storage to work around the failure until it has been resolved. If the drive is going to be replaced immediately, then it is just best to replace the drive, format it, remount it, and let replication fill it up.

If you cannot replace the drive immediately, then it is best to leave it unmounted, and remove the drive from the ring. This will allow all the replicas that were on that drive to be replicated elsewhere until the drive is replaced. Once the drive is replaced, it can be re-added to the ring.

You can look at error messages in the `/var/log/kern.log` file for hints of drive failure.

Server failure

Problem

The server is potentially offline, and may have failed, or require a reboot.

Solution

If a server is having hardware issues, it is a good idea to make sure the Object Storage services are not running. This will allow Object Storage to work around the failure while you troubleshoot.

If the server just needs a reboot, or a small amount of work that should only last a couple of hours, then it is probably best to let Object Storage work around the failure and get the machine fixed and back online. When the machine comes back online, replication will make sure that anything that is missing during the downtime will get updated.

If the server has more serious issues, then it is probably best to remove all of the servers devices from the ring. Once the server has been repaired and is back online, the servers devices can be added back into the ring. It is important that the devices are reformatted before putting them back into the ring as it is likely to be responsible for a different set of partitions than before.

Detect failed drives

Problem

When drives fail, it can be difficult to detect that a drive has failed, and the details of the failure.

Solution

It has been our experience that when a drive is about to fail, error messages appear in the `/var/log/kern.log` file. There is a script called `swift-drive-audit` that can be run via cron to watch for bad drives. If errors are detected, it will unmount the bad drive, so that Object Storage can work around it. The script takes a configuration file with the following settings:

Table 1: **Description of configuration options for [drive-audit] in drive-audit.conf**

Configuration option = Default value	Description
device_dir = /srv/node	Directory devices are mounted under
error_limit = 1	Number of errors to find before a device is unmounted
log_address = /dev/log	Location where syslog sends the logs to
log_facility = LOG_LOCAL0	Syslog log facility
log_file_pattern = /var/log/kern.*[!.][!g][!z]	Location of the log file with globbing pattern to check against device errors locate device blocks with errors in the log file
log_level = INFO	Logging level
log_max_line_length = 0	Caps the length of log lines to the value given; no limit if set to 0, the default.
log_to_console = False	No help text available for this option.
minutes = 60	Number of minutes to look back in /var/log/kern.log
recon_cache_path = /var/cache/swift	Directory where stats for a few items will be stored
regex_pattern_1 = \berror\b.*\b(dm-[0-9]{1,2}\d?)\b	No help text available for this option.
unmount_failed_device = True	No help text available for this option.

Warning: This script has only been tested on Ubuntu 10.04; use with caution on other operating systems in production.

Emergency recovery of ring builder files

Problem

An emergency might prevent a successful backup from restoring the cluster to operational status.

Solution

You should always keep a backup of swift ring builder files. However, if an emergency occurs, this procedure may assist in returning your cluster to an operational state.

Using existing swift tools, there is no way to recover a builder file from a `ring.gz` file. However, if you have a knowledge of Python, it is possible to construct a builder file that is pretty close to the one you have lost.

Warning: This procedure is a last-resort for emergency circumstances. It requires knowledge of the swift python code and may not succeed.

1. Load the ring and a new ringbuilder object in a Python REPL:

```
>>> from swift.common.ring import RingData, RingBuilder
>>> ring = RingData.load('/path/to/account.ring.gz')
```

2. Start copying the data we have in the ring into the builder:

```
>>> import math
>>> partitions = len(ring._replica2part2dev_id[0])
>>> replicas = len(ring._replica2part2dev_id)

>>> builder = RingBuilder(int(math.log(partitions, 2)), replicas, 1)
>>> builder.devs = ring.devs
>>> builder._replica2part2dev = ring._replica2part2dev_id
>>> builder._last_part_moves_epoch = 0
>>> from array import array
>>> builder._last_part_moves = array('B', (0 for _ in range(partitions)))
>>> builder._set_parts_wanted()
>>> for d in builder._iter_devs():
            d['parts'] = 0
>>> for p2d in builder._replica2part2dev:
            for dev_id in p2d:
                builder.devs[dev_id]['parts'] += 1

This is the extent of the recoverable fields.
```

3. For `min_part_hours` you either have to remember what the value you used was, or just make up a new one:

```
>>> builder.change_min_part_hours(24) # or whatever you want it to be
```

4. Validate the builder. If this raises an exception, check your previous code:

```
>>> builder.validate()
```

5. After it validates, save the builder and create a new `account.builder`:

```
>>> import pickle
>>> pickle.dump(builder.to_dict(), open('account.builder', 'wb'),␣
↪protocol=2)
>>> exit ()
```

6. You should now have a file called `account.builder` in the current working directory. Run **`swift-ring-builder account.builder write_ring`** and compare the new `account.ring.gz` to the `account.ring.gz` that you started from. They probably are not byte-for-byte identical, but if you load them in a REPL and their `_replica2part2dev_id` and `devs` attributes are the same (or nearly so), then you are in good shape.

7. Repeat the procedure for `container.ring.gz` and `object.ring.gz`, and you might get usable builder files.

5.9 Object Storage Install Guide

5.9.1 Object Storage service overview

The OpenStack Object Storage is a multi-tenant object storage system. It is highly scalable and can manage large amounts of unstructured data at low cost through a RESTful HTTP API.

It includes the following components:

Proxy servers (swift-proxy-server) Accepts OpenStack Object Storage API and raw HTTP requests to upload files, modify metadata, and create containers. It also serves file or container listings to web browsers. To improve performance, the proxy server can use an optional cache that is usually deployed with memcache.

Account servers (swift-account-server) Manages accounts defined with Object Storage.

Container servers (swift-container-server) Manages the mapping of containers or folders, within Object Storage.

Object servers (swift-object-server) Manages actual objects, such as files, on the storage nodes.

Various periodic processes Performs housekeeping tasks on the large data store. The replication services ensure consistency and availability through the cluster. Other periodic processes include auditors, updaters, and reapers.

WSGI middleware Handles authentication and is usually OpenStack Identity.

swift client Enables users to submit commands to the REST API through a command-line client authorized as either a admin user, reseller user, or swift user.

swift-init Script that initializes the building of the ring file, takes daemon names as parameter and offers commands. Documented in https://docs.openstack.org/swift/latest/admin_guide.html#managing-services.

swift-recon A cli tool used to retrieve various metrics and telemetry information about a cluster that has been collected by the swift-recon middleware.

swift-ring-builder Storage ring build and rebalance utility. Documented in https://docs.openstack.org/swift/latest/admin_guide.html#managing-the-rings.

5.9.2 Configure networking

Before you start deploying the Object Storage service in your OpenStack environment, configure networking for two additional storage nodes.

First node

Configure network interfaces

- Configure the management interface:
  - IP address: `10.0.0.51`
  - Network mask: `255.255.255.0` (or `/24`)
  - Default gateway: `10.0.0.1`

Configure name resolution

1. Set the hostname of the node to `object1`.
2. Edit the `/etc/hosts` file to contain the following:

```
# controller
10.0.0.11       controller

# compute1
10.0.0.31       compute1

# block1
10.0.0.41       block1

# object1
10.0.0.51       object1

# object2
10.0.0.52       object2
```

3. Reboot the system to activate the changes.

Second node

Configure network interfaces

- Configure the management interface:
  - IP address: `10.0.0.52`
  - Network mask: `255.255.255.0` (or `/24`)
  - Default gateway: `10.0.0.1`

Configure name resolution

1. Set the hostname of the node to `object2`.
2. Edit the `/etc/hosts` file to contain the following:

```
# controller
10.0.0.11       controller

# compute1
10.0.0.31       compute1

# block1
10.0.0.41       block1

# object1
```

(continues on next page)

(continued from previous page)

```
10.0.0.51       object1

# object2
10.0.0.52       object2
```

3. Reboot the system to activate the changes.

Warning: Some distributions add an extraneous entry in the `/etc/hosts` file that resolves the actual hostname to another loopback IP address such as `127.0.1.1`. You must comment out or remove this entry to prevent name resolution problems. **Do not remove the 127.0.0.1 entry.**

Note: To reduce complexity of this guide, we add host entries for optional services regardless of whether you choose to deploy them.

5.9.3 Install and configure the controller node

This section describes how to install and configure the proxy service that handles requests for the account, container, and object services operating on the storage nodes.

Note that installation and configuration vary by distribution.

Install and configure the controller node for openSUSE and SUSE Linux Enterprise

This section describes how to install and configure the proxy service that handles requests for the account, container, and object services operating on the storage nodes. For simplicity, this guide installs and configures the proxy service on the controller node. However, you can run the proxy service on any node with network connectivity to the storage nodes. Additionally, you can install and configure the proxy service on multiple nodes to increase performance and redundancy. For more information, see the Deployment Guide.

This section applies to openSUSE Leap 42.2 and SUSE Linux Enterprise Server 12 SP2.

Prerequisites

The proxy service relies on an authentication and authorization mechanism such as the Identity service. However, unlike other services, it also offers an internal mechanism that allows it to operate without any other OpenStack services. Before you configure the Object Storage service, you must create service credentials and an API endpoint.

Note: The Object Storage service does not use an SQL database on the controller node. Instead, it uses distributed SQLite databases on each storage node.

1. Source the `admin` credentials to gain access to admin-only CLI commands:

```
$ . admin-openrc
```

2. To create the Identity service credentials, complete these steps:

   - Create the `swift` user:

```
$ openstack user create --domain default --password-prompt swift
User Password:
Repeat User Password:
+-----------+----------------------------------+
| Field     | Value                            |
+-----------+----------------------------------+
| domain_id | default                          |
| enabled   | True                             |
| id        | d535e5cbd2b74ac7bfb97db9cced3ed6 |
| name      | swift                            |
+-----------+----------------------------------+
```

   - Add the `admin` role to the `swift` user:

```
$ openstack role add --project service --user swift admin
```

   Note: This command provides no output.

   - Create the `swift` service entity:

```
$ openstack service create --name swift \
  --description "OpenStack Object Storage" object-store
+-------------+----------------------------------+
| Field       | Value                            |
+-------------+----------------------------------+
| description | OpenStack Object Storage         |
| enabled     | True                             |
| id          | 75ef509da2c340499d454ae96a2c5c34 |
| name        | swift                            |
| type        | object-store                     |
+-------------+----------------------------------+
```

3. Create the Object Storage service API endpoints:

```
$ openstack endpoint create --region RegionOne \
  object-store public http://controller:8080/v1/AUTH_%\(project_id\)s
+--------------+------------------------------------------------+
| Field        | Value                                          |
+--------------+------------------------------------------------+
| enabled      | True                                           |
| id           | 12bfd36f26694c97813f665707114e0d               |
| interface    | public                                         |
| region       | RegionOne                                      |
| region_id    | RegionOne                                      |
```

(continues on next page)

(continued from previous page)

```
| service_id   | 75ef509da2c340499d454ae96a2c5c34              |
| service_name | swift                                         |
| service_type | object-store                                  |
| url          | http://controller:8080/v1/AUTH_%(project_id)s |
+--------------+-----------------------------------------------+

$ openstack endpoint create --region RegionOne \
  object-store internal http://controller:8080/v1/AUTH_%\(project_id\)s
+--------------+-----------------------------------------------+
| Field        | Value                                         |
+--------------+-----------------------------------------------+
| enabled      | True                                          |
| id           | 7a36bee6733a4b5590d74d3080ee6789              |
| interface    | internal                                      |
| region       | RegionOne                                     |
| region_id    | RegionOne                                     |
| service_id   | 75ef509da2c340499d454ae96a2c5c34              |
| service_name | swift                                         |
| service_type | object-store                                  |
| url          | http://controller:8080/v1/AUTH_%(project_id)s |
+--------------+-----------------------------------------------+

$ openstack endpoint create --region RegionOne \
  object-store admin http://controller:8080/v1
+--------------+----------------------------------+
| Field        | Value                            |
+--------------+----------------------------------+
| enabled      | True                             |
| id           | ebb72cd6851d4defabc0b9d71cdca69b |
| interface    | admin                            |
| region       | RegionOne                        |
| region_id    | RegionOne                        |
| service_id   | 75ef509da2c340499d454ae96a2c5c34 |
| service_name | swift                            |
| service_type | object-store                     |
| url          | http://controller:8080/v1        |
+--------------+----------------------------------+
```

Install and configure components

Note: Default configuration files vary by distribution. You might need to add these sections and options rather than modifying existing sections and options. Also, an ellipsis (`...`) in the configuration snippets indicates potential default configuration options that you should retain.

1. Install the packages:

```
# zypper install openstack-swift-proxy python-swiftclient \
  python-keystoneclient python-keystonemiddleware \
  python-xml memcached
```

Note: Complete OpenStack environments already include some of these packages.

2. Edit the `/etc/swift/proxy-server.conf` file and complete the following actions:

   - In the `[DEFAULT]` section, configure the bind port, user, and configuration directory:

     ```
     [DEFAULT]
     ...
     bind_port = 8080
     user = swift
     swift_dir = /etc/swift
     ```

   - In the `[pipeline:main]` section, remove the `tempurl` and `tempauth` modules and add the `authtoken` and `keystoneauth` modules:

     ```
     [pipeline:main]
     pipeline = catch_errors gatekeeper healthcheck proxy-logging␣
      ↪cache container_sync bulk ratelimit authtoken keystoneauth␣
      ↪container-quotas account-quotas slo dlo versioned_writes proxy-
      ↪logging proxy-server
     ```

     Note: Do not change the order of the modules.

     Note: For more information on other modules that enable additional features, see the Deployment Guide.

   - In the `[app:proxy-server]` section, enable automatic account creation:

     ```
     [app:proxy-server]
     use = egg:swift#proxy
     ...
     account_autocreate = True
     ```

   - In the `[filter:keystoneauth]` section, configure the operator roles:

     ```
     [filter:keystoneauth]
     use = egg:swift#keystoneauth
     ...
     operator_roles = admin,user
     ```

   - In the `[filter:authtoken]` section, configure Identity service access:

```
[filter:authtoken]
paste.filter_factory = keystonemiddleware.auth_token:filter_
↪factory
...
www_authenticate_uri = http://controller:5000
auth_url = http://controller:5000
memcached_servers = controller:11211
auth_type = password
project_domain_id = default
user_domain_id = default
project_name = service
username = swift
password = SWIFT_PASS
delay_auth_decision = True
```

Replace `SWIFT_PASS` with the password you chose for the `swift` user in the Identity service.

Note: Comment out or remove any other options in the `[filter:authtoken]` section.

- In the `[filter:cache]` section, configure the `memcached` location:

```
[filter:cache]
use = egg:swift#memcache
...
memcache_servers = controller:11211
```

Install and configure the controller node for Red Hat Enterprise Linux and CentOS

This section describes how to install and configure the proxy service that handles requests for the account, container, and object services operating on the storage nodes. For simplicity, this guide installs and configures the proxy service on the controller node. However, you can run the proxy service on any node with network connectivity to the storage nodes. Additionally, you can install and configure the proxy service on multiple nodes to increase performance and redundancy. For more information, see the Deployment Guide.

This section applies to Red Hat Enterprise Linux 7 and CentOS 7.

Prerequisites

The proxy service relies on an authentication and authorization mechanism such as the Identity service. However, unlike other services, it also offers an internal mechanism that allows it to operate without any other OpenStack services. Before you configure the Object Storage service, you must create service credentials and an API endpoint.

Note: The Object Storage service does not use an SQL database on the controller node. Instead, it uses distributed SQLite databases on each storage node.

1. Source the `admin` credentials to gain access to admin-only CLI commands:

```
$ . admin-openrc
```

2. To create the Identity service credentials, complete these steps:

   - Create the `swift` user:

```
$ openstack user create --domain default --password-prompt swift
User Password:
Repeat User Password:
+-----------+----------------------------------+
| Field     | Value                            |
+-----------+----------------------------------+
| domain_id | default                          |
| enabled   | True                             |
| id        | d535e5cbd2b74ac7bfb97db9cced3ed6 |
| name      | swift                            |
+-----------+----------------------------------+
```

   - Add the `admin` role to the `swift` user:

```
$ openstack role add --project service --user swift admin
```

Note: This command provides no output.

   - Create the `swift` service entity:

```
$ openstack service create --name swift \
  --description "OpenStack Object Storage" object-store
+-------------+----------------------------------+
| Field       | Value                            |
+-------------+----------------------------------+
| description | OpenStack Object Storage         |
| enabled     | True                             |
| id          | 75ef509da2c340499d454ae96a2c5c34 |
| name        | swift                            |
| type        | object-store                     |
+-------------+----------------------------------+
```

3. Create the Object Storage service API endpoints:

```
$ openstack endpoint create --region RegionOne \
  object-store public http://controller:8080/v1/AUTH_%\(project_id\)s
+--------------+------------------------------------------+
| Field        | Value                                    |
+--------------+------------------------------------------+
| enabled      | True                                     |
| id           | 12bfd36f26694c97813f665707114e0d         |
| interface    | public                                   |
| region       | RegionOne                                |
```

(continues on next page)

(continued from previous page)

```
| region_id    | RegionOne                                       |
| service_id   | 75ef509da2c340499d454ae96a2c5c34                |
| service_name | swift                                           |
| service_type | object-store                                    |
| url          | http://controller:8080/v1/AUTH_%(project_id)s |
+--------------+-------------------------------------------------+

$ openstack endpoint create --region RegionOne \
  object-store internal http://controller:8080/v1/AUTH_%\(project_id\)s
+--------------+-------------------------------------------------+
| Field        | Value                                           |
+--------------+-------------------------------------------------+
| enabled      | True                                            |
| id           | 7a36bee6733a4b5590d74d3080ee6789                |
| interface    | internal                                        |
| region       | RegionOne                                       |
| region_id    | RegionOne                                       |
| service_id   | 75ef509da2c340499d454ae96a2c5c34                |
| service_name | swift                                           |
| service_type | object-store                                    |
| url          | http://controller:8080/v1/AUTH_%(project_id)s |
+--------------+-------------------------------------------------+

$ openstack endpoint create --region RegionOne \
  object-store admin http://controller:8080/v1
+--------------+----------------------------------+
| Field        | Value                            |
+--------------+----------------------------------+
| enabled      | True                             |
| id           | ebb72cd6851d4defabc0b9d71cdca69b |
| interface    | admin                            |
| region       | RegionOne                        |
| region_id    | RegionOne                        |
| service_id   | 75ef509da2c340499d454ae96a2c5c34 |
| service_name | swift                            |
| service_type | object-store                     |
| url          | http://controller:8080/v1        |
+--------------+----------------------------------+
```

Install and configure components

Note: Default configuration files vary by distribution. You might need to add these sections and options rather than modifying existing sections and options. Also, an ellipsis (`...`) in the configuration snippets indicates potential default configuration options that you should retain.

1. Install the packages:

```
# yum install openstack-swift-proxy python-swiftclient \
  python-keystoneclient python-keystonemiddleware \
  memcached
```

Note: Complete OpenStack environments already include some of these packages.

2. Obtain the proxy service configuration file from the Object Storage source repository:

```
# curl -o /etc/swift/proxy-server.conf https://opendev.org/openstack/
↪swift/raw/branch/master/etc/proxy-server.conf-sample
```

3. Edit the `/etc/swift/proxy-server.conf` file and complete the following actions:

   - In the `[DEFAULT]` section, configure the bind port, user, and configuration directory:

```
[DEFAULT]
...
bind_port = 8080
user = swift
swift_dir = /etc/swift
```

   - In the `[pipeline:main]` section, remove the `tempurl` and `tempauth` modules and add the `authtoken` and `keystoneauth` modules:

```
[pipeline:main]
pipeline = catch_errors gatekeeper healthcheck proxy-logging␣
↪cache container_sync bulk ratelimit authtoken keystoneauth␣
↪container-quotas account-quotas slo dlo versioned_writes proxy-
↪logging proxy-server
```

     Note: Do not change the order of the modules.

     Note: For more information on other modules that enable additional features, see the Deployment Guide.

   - In the `[app:proxy-server]` section, enable automatic account creation:

```
[app:proxy-server]
use = egg:swift#proxy
...
account_autocreate = True
```

   - In the `[filter:keystoneauth]` section, configure the operator roles:

```
[filter:keystoneauth]
use = egg:swift#keystoneauth
...
operator_roles = admin,user
```

- In the [filter:authtoken] section, configure Identity service access:

```
[filter:authtoken]
paste.filter_factory = keystonemiddleware.auth_token:filter_
↪factory
...
www_authenticate_uri = http://controller:5000
auth_url = http://controller:5000
memcached_servers = controller:11211
auth_type = password
project_domain_id = default
user_domain_id = default
project_name = service
username = swift
password = SWIFT_PASS
delay_auth_decision = True
```

Replace SWIFT_PASS with the password you chose for the swift user in the Identity service.

Note: Comment out or remove any other options in the [filter:authtoken] section.

- In the [filter:cache] section, configure the memcached location:

```
[filter:cache]
use = egg:swift#memcache
...
memcache_servers = controller:11211
```

Install and configure the controller node for Ubuntu

This section describes how to install and configure the proxy service that handles requests for the account, container, and object services operating on the storage nodes. For simplicity, this guide installs and configures the proxy service on the controller node. However, you can run the proxy service on any node with network connectivity to the storage nodes. Additionally, you can install and configure the proxy service on multiple nodes to increase performance and redundancy. For more information, see the Deployment Guide.

This section applies to Ubuntu 14.04 (LTS).

Prerequisites

The proxy service relies on an authentication and authorization mechanism such as the Identity service. However, unlike other services, it also offers an internal mechanism that allows it to operate without any other OpenStack services. Before you configure the Object Storage service, you must create service credentials and an API endpoint.

Note: The Object Storage service does not use an SQL database on the controller node. Instead, it uses

distributed SQLite databases on each storage node.

1. Source the `admin` credentials to gain access to admin-only CLI commands:

```
$ . admin-openrc
```

2. To create the Identity service credentials, complete these steps:

   - Create the `swift` user:

```
$ openstack user create --domain default --password-prompt swift
User Password:
Repeat User Password:
+-----------+----------------------------------+
| Field     | Value                            |
+-----------+----------------------------------+
| domain_id | default                          |
| enabled   | True                             |
| id        | d535e5cbd2b74ac7bfb97db9cced3ed6 |
| name      | swift                            |
+-----------+----------------------------------+
```

   - Add the `admin` role to the `swift` user:

```
$ openstack role add --project service --user swift admin
```

   Note: This command provides no output.

   - Create the `swift` service entity:

```
$ openstack service create --name swift \
  --description "OpenStack Object Storage" object-store
+-------------+----------------------------------+
| Field       | Value                            |
+-------------+----------------------------------+
| description | OpenStack Object Storage         |
| enabled     | True                             |
| id          | 75ef509da2c340499d454ae96a2c5c34 |
| name        | swift                            |
| type        | object-store                     |
+-------------+----------------------------------+
```

3. Create the Object Storage service API endpoints:

```
$ openstack endpoint create --region RegionOne \
  object-store public http://controller:8080/v1/AUTH_%\(project_id\)s
+--------------+------------------------------------------+
| Field        | Value                                    |
+--------------+------------------------------------------+
| enabled      | True                                     |
```

(continues on next page)

(continued from previous page)

```
| id           | 12bfd36f26694c97813f665707114e0d              |
| interface    | public                                        |
| region       | RegionOne                                     |
| region_id    | RegionOne                                     |
| service_id   | 75ef509da2c340499d454ae96a2c5c34              |
| service_name | swift                                         |
| service_type | object-store                                  |
| url          | http://controller:8080/v1/AUTH_%(project_id)s |
+--------------+-----------------------------------------------+

$ openstack endpoint create --region RegionOne \
  object-store internal http://controller:8080/v1/AUTH_%\(project_id\)s
+--------------+-----------------------------------------------+
| Field        | Value                                         |
+--------------+-----------------------------------------------+
| enabled      | True                                          |
| id           | 7a36bee6733a4b5590d74d3080ee6789              |
| interface    | internal                                      |
| region       | RegionOne                                     |
| region_id    | RegionOne                                     |
| service_id   | 75ef509da2c340499d454ae96a2c5c34              |
| service_name | swift                                         |
| service_type | object-store                                  |
| url          | http://controller:8080/v1/AUTH_%(project_id)s |
+--------------+-----------------------------------------------+

$ openstack endpoint create --region RegionOne \
  object-store admin http://controller:8080/v1
+--------------+----------------------------------+
| Field        | Value                            |
+--------------+----------------------------------+
| enabled      | True                             |
| id           | ebb72cd6851d4defabc0b9d71cdca69b |
| interface    | admin                            |
| region       | RegionOne                        |
| region_id    | RegionOne                        |
| service_id   | 75ef509da2c340499d454ae96a2c5c34 |
| service_name | swift                            |
| service_type | object-store                     |
| url          | http://controller:8080/v1        |
+--------------+----------------------------------+
```

Install and configure components

Note: Default configuration files vary by distribution. You might need to add these sections and options rather than modifying existing sections and options. Also, an ellipsis (`...`) in the configuration snippets indicates potential default configuration options that you should retain.

1. Install the packages:

```
# apt-get install swift swift-proxy python-swiftclient \
  python-keystoneclient python-keystonemiddleware \
  memcached
```

Note: Complete OpenStack environments already include some of these packages.

2. Create the `/etc/swift` directory.
3. Obtain the proxy service configuration file from the Object Storage source repository:

```
# curl -o /etc/swift/proxy-server.conf https://opendev.org/openstack/
↪swift/raw/branch/master/etc/proxy-server.conf-sample
```

4. Edit the `/etc/swift/proxy-server.conf` file and complete the following actions:
   - In the `[DEFAULT]` section, configure the bind port, user, and configuration directory:

```
[DEFAULT]
...
bind_port = 8080
user = swift
swift_dir = /etc/swift
```

   - In the `[pipeline:main]` section, remove the `tempurl` and `tempauth` modules and add the `authtoken` and `keystoneauth` modules:

```
[pipeline:main]
pipeline = catch_errors gatekeeper healthcheck proxy-logging␣
↪cache container_sync bulk ratelimit authtoken keystoneauth␣
↪container-quotas account-quotas slo dlo versioned_writes proxy-
↪logging proxy-server
```

Note: Do not change the order of the modules.

Note: For more information on other modules that enable additional features, see the Deployment Guide.

   - In the `[app:proxy-server]` section, enable automatic account creation:

```
[app:proxy-server]
use = egg:swift#proxy
...
account_autocreate = True
```

- In the `[filter:keystoneauth]` section, configure the operator roles:

```
[filter:keystoneauth]
use = egg:swift#keystoneauth
...
operator_roles = admin,user
```

- In the `[filter:authtoken]` section, configure Identity service access:

```
[filter:authtoken]
paste.filter_factory = keystonemiddleware.auth_token:filter_
↪factory
...
www_authenticate_uri = http://controller:5000
auth_url = http://controller:5000
memcached_servers = controller:11211
auth_type = password
project_domain_id = default
user_domain_id = default
project_name = service
username = swift
password = SWIFT_PASS
delay_auth_decision = True
```

Replace `SWIFT_PASS` with the password you chose for the `swift` user in the Identity service.

Note: Comment out or remove any other options in the `[filter:authtoken]` section.

- In the `[filter:cache]` section, configure the `memcached` location:

```
[filter:cache]
use = egg:swift#memcache
...
memcache_servers = controller:11211
```

Install and configure the controller node for Debian

This section describes how to install and configure the proxy service that handles requests for the account, container, and object services operating on the storage nodes. For simplicity, this guide installs and configures the proxy service on the controller node. However, you can run the proxy service on any node with network connectivity to the storage nodes. Additionally, you can install and configure the proxy service on multiple nodes to increase performance and redundancy. For more information, see the Deployment Guide.

This section applies to Debian.

Prerequisites

The proxy service relies on an authentication and authorization mechanism such as the Identity service. However, unlike other services, it also offers an internal mechanism that allows it to operate without any other OpenStack services. Before you configure the Object Storage service, you must create service credentials and an API endpoint.

Note: The Object Storage service does not use an SQL database on the controller node. Instead, it uses distributed SQLite databases on each storage node.

---

1. Source the `admin` credentials to gain access to admin-only CLI commands:

```
$ . admin-openrc
```

2. To create the Identity service credentials, complete these steps:

   - Create the `swift` user:

```
$ openstack user create --domain default --password-prompt swift
User Password:
Repeat User Password:
+-----------+----------------------------------+
| Field     | Value                            |
+-----------+----------------------------------+
| domain_id | default                          |
| enabled   | True                             |
| id        | d535e5cbd2b74ac7bfb97db9cced3ed6 |
| name      | swift                            |
+-----------+----------------------------------+
```

   - Add the `admin` role to the `swift` user:

```
$ openstack role add --project service --user swift admin
```

Note: This command provides no output.

   - Create the `swift` service entity:

```
$ openstack service create --name swift \
  --description "OpenStack Object Storage" object-store
+-------------+----------------------------------+
| Field       | Value                            |
+-------------+----------------------------------+
| description | OpenStack Object Storage         |
| enabled     | True                             |
| id          | 75ef509da2c340499d454ae96a2c5c34 |
| name        | swift                            |
| type        | object-store                     |
+-------------+----------------------------------+
```

3. Create the Object Storage service API endpoints:

```
$ openstack endpoint create --region RegionOne \
  object-store public http://controller:8080/v1/AUTH_%\(project_id\)s
+--------------+-------------------------------------------------+
| Field        | Value                                           |
+--------------+-------------------------------------------------+
| enabled      | True                                            |
| id           | 12bfd36f26694c97813f665707114e0d                |
| interface    | public                                          |
| region       | RegionOne                                       |
| region_id    | RegionOne                                       |
| service_id   | 75ef509da2c340499d454ae96a2c5c34                |
| service_name | swift                                           |
| service_type | object-store                                    |
| url          | http://controller:8080/v1/AUTH_%(project_id)s |
+--------------+-------------------------------------------------+

$ openstack endpoint create --region RegionOne \
  object-store internal http://controller:8080/v1/AUTH_%\(project_id\)s
+--------------+-------------------------------------------------+
| Field        | Value                                           |
+--------------+-------------------------------------------------+
| enabled      | True                                            |
| id           | 7a36bee6733a4b5590d74d3080ee6789                |
| interface    | internal                                        |
| region       | RegionOne                                       |
| region_id    | RegionOne                                       |
| service_id   | 75ef509da2c340499d454ae96a2c5c34                |
| service_name | swift                                           |
| service_type | object-store                                    |
| url          | http://controller:8080/v1/AUTH_%(project_id)s |
+--------------+-------------------------------------------------+

$ openstack endpoint create --region RegionOne \
  object-store admin http://controller:8080/v1
+--------------+----------------------------------+
| Field        | Value                            |
```

(continues on next page)

(continued from previous page)

```
+--------------+----------------------------------+
| enabled      | True                             |
| id           | ebb72cd6851d4defabc0b9d71cdca69b |
| interface    | admin                            |
| region       | RegionOne                        |
| region_id    | RegionOne                        |
| service_id   | 75ef509da2c340499d454ae96a2c5c34 |
| service_name | swift                            |
| service_type | object-store                     |
| url          | http://controller:8080/v1        |
+--------------+----------------------------------+
```

Install and configure components

Note: Default configuration files vary by distribution. You might need to add these sections and options rather than modifying existing sections and options. Also, an ellipsis (`...`) in the configuration snippets indicates potential default configuration options that you should retain.

1. Install the packages:

```
# apt-get install swift swift-proxy python-swiftclient \
  python-keystoneclient python-keystonemiddleware \
  memcached
```

Note: Complete OpenStack environments already include some of these packages.

2. Create the `/etc/swift` directory.
3. Obtain the proxy service configuration file from the Object Storage source repository:

```
# curl -o /etc/swift/proxy-server.conf https://opendev.org/openstack/
↪swift/raw/branch/master/etc/proxy-server.conf-sample
```

4. Edit the `/etc/swift/proxy-server.conf` file and complete the following actions:
   - In the `[DEFAULT]` section, configure the bind port, user, and configuration directory:

```
[DEFAULT]
...
bind_port = 8080
user = swift
swift_dir = /etc/swift
```

   - In the `[pipeline:main]` section, remove the `tempurl` and `tempauth` modules and add the `authtoken` and `keystoneauth` modules:

```
[pipeline:main]
pipeline = catch_errors gatekeeper healthcheck proxy-logging␣
↪cache container_sync bulk ratelimit authtoken keystoneauth␣
↪container-quotas account-quotas slo dlo versioned_writes proxy-
↪logging proxy-server
```

Note: Do not change the order of the modules.

Note: For more information on other modules that enable additional features, see the Deployment Guide.

- In the `[app:proxy-server]` section, enable automatic account creation:

```
[app:proxy-server]
use = egg:swift#proxy
...
account_autocreate = True
```

- In the `[filter:keystoneauth]` section, configure the operator roles:

```
[filter:keystoneauth]
use = egg:swift#keystoneauth
...
operator_roles = admin,user
```

- In the `[filter:authtoken]` section, configure Identity service access:

```
[filter:authtoken]
paste.filter_factory = keystonemiddleware.auth_token:filter_
↪factory
...
www_authenticate_uri = http://controller:5000
auth_url = http://controller:5000
memcached_servers = controller:11211
auth_type = password
project_domain_id = default
user_domain_id = default
project_name = service
username = swift
password = SWIFT_PASS
delay_auth_decision = True
```

Replace `SWIFT_PASS` with the password you chose for the `swift` user in the Identity service.

Note: Comment out or remove any other options in the `[filter:authtoken]` section.

- In the `[filter:cache]` section, configure the `memcached` location:

```
[filter:cache]
use = egg:swift#memcache
...
memcache_servers = controller:11211
```

5.9.4 Install and configure the storage nodes

This section describes how to install and configure storage nodes that operate the account, container, and object services.

Note that installation and configuration vary by distribution.

Install and configure the storage nodes for openSUSE and SUSE Linux Enterprise

This section describes how to install and configure storage nodes that operate the account, container, and object services. For simplicity, this configuration references two storage nodes, each containing two empty local block storage devices. The instructions use `/dev/sdb` and `/dev/sdc`, but you can substitute different values for your particular nodes.

Although Object Storage supports any file system with extended attributes (xattr), testing and benchmarking indicate the best performance and reliability on XFS. For more information on horizontally scaling your environment, see the Deployment Guide.

This section applies to openSUSE Leap 42.2 and SUSE Linux Enterprise Server 12 SP2.

Prerequisites

Before you install and configure the Object Storage service on the storage nodes, you must prepare the storage devices.

Note: Perform these steps on each storage node.

1. Install the supporting utility packages:

```
# zypper install xfsprogs rsync
```

2. Format the `/dev/sdb` and `/dev/sdc` devices as XFS:

```
# mkfs.xfs /dev/sdb
# mkfs.xfs /dev/sdc
```

3. Create the mount point directory structure:

```
# mkdir -p /srv/node/sdb
# mkdir -p /srv/node/sdc
```

4. Find the UUID of the new partitions:

```
# blkid
```

5. Edit the `/etc/fstab` file and add the following to it:

```
UUID="<UUID-from-output-above>" /srv/node/sdb xfs noatime 0 2
UUID="<UUID-from-output-above>" /srv/node/sdc xfs noatime 0 2
```

6. Mount the devices:

```
# mount /srv/node/sdb
# mount /srv/node/sdc
```

7. Create or edit the `/etc/rsyncd.conf` file to contain the following:

```
uid = swift
gid = swift
log file = /var/log/rsyncd.log
pid file = /var/run/rsyncd.pid
address = MANAGEMENT_INTERFACE_IP_ADDRESS

[account]
max connections = 2
path = /srv/node/
read only = False
lock file = /var/lock/account.lock

[container]
max connections = 2
path = /srv/node/
read only = False
lock file = /var/lock/container.lock

[object]
max connections = 2
path = /srv/node/
read only = False
lock file = /var/lock/object.lock
```

Replace `MANAGEMENT_INTERFACE_IP_ADDRESS` with the IP address of the management network on the storage node.

Note: The `rsync` service requires no authentication, so consider running it on a private network in production environments.

---

7. Start the `rsyncd` service and configure it to start when the system boots:

```
# systemctl enable rsyncd.service
# systemctl start rsyncd.service
```

Install and configure components

Note: Default configuration files vary by distribution. You might need to add these sections and options rather than modifying existing sections and options. Also, an ellipsis (`...`) in the configuration snippets indicates potential default configuration options that you should retain.

Note: Perform these steps on each storage node.

1. Install the packages:

```
# zypper install openstack-swift-account \
  openstack-swift-container openstack-swift-object python-xml
```

2. Edit the `/etc/swift/account-server.conf` file and complete the following actions:

   - In the `[DEFAULT]` section, configure the bind IP address, bind port, user, configuration directory, and mount point directory:

```
[DEFAULT]
...
bind_ip = MANAGEMENT_INTERFACE_IP_ADDRESS
bind_port = 6202
user = swift
swift_dir = /etc/swift
devices = /srv/node
mount_check = True
```

     Replace `MANAGEMENT_INTERFACE_IP_ADDRESS` with the IP address of the management network on the storage node.

   - In the `[pipeline:main]` section, enable the appropriate modules:

```
[pipeline:main]
pipeline = healthcheck recon account-server
```

     Note: For more information on other modules that enable additional features, see the Deployment Guide.

   - In the `[filter:recon]` section, configure the recon (meters) cache directory:

```
[filter:recon]
use = egg:swift#recon
...
recon_cache_path = /var/cache/swift
```

3. Edit the `/etc/swift/container-server.conf` file and complete the following actions:

   - In the `[DEFAULT]` section, configure the bind IP address, bind port, user, configuration directory, and mount point directory:

```
[DEFAULT]
...
bind_ip = MANAGEMENT_INTERFACE_IP_ADDRESS
bind_port = 6201
user = swift
swift_dir = /etc/swift
devices = /srv/node
mount_check = True
```

Replace `MANAGEMENT_INTERFACE_IP_ADDRESS` with the IP address of the management network on the storage node.

- In the `[pipeline:main]` section, enable the appropriate modules:

```
[pipeline:main]
pipeline = healthcheck recon container-server
```

Note: For more information on other modules that enable additional features, see the Deployment Guide.

- In the `[filter:recon]` section, configure the recon (meters) cache directory:

```
[filter:recon]
use = egg:swift#recon
...
recon_cache_path = /var/cache/swift
```

4. Edit the `/etc/swift/object-server.conf` file and complete the following actions:

   - In the `[DEFAULT]` section, configure the bind IP address, bind port, user, configuration directory, and mount point directory:

```
[DEFAULT]
...
bind_ip = MANAGEMENT_INTERFACE_IP_ADDRESS
bind_port = 6200
user = swift
swift_dir = /etc/swift
devices = /srv/node
mount_check = True
```

   Replace `MANAGEMENT_INTERFACE_IP_ADDRESS` with the IP address of the management network on the storage node.

   - In the `[pipeline:main]` section, enable the appropriate modules:

```
[pipeline:main]
pipeline = healthcheck recon object-server
```

Note: For more information on other modules that enable additional features, see the De-

ployment Guide.

- In the `[filter:recon]` section, configure the recon (meters) cache and lock directories:

```
[filter:recon]
use = egg:swift#recon
...
recon_cache_path = /var/cache/swift
recon_lock_path = /var/lock
```

5. Ensure proper ownership of the mount point directory structure:

```
# chown -R swift:swift /srv/node
```

Install and configure the storage nodes for Red Hat Enterprise Linux and CentOS

This section describes how to install and configure storage nodes that operate the account, container, and object services. For simplicity, this configuration references two storage nodes, each containing two empty local block storage devices. The instructions use `/dev/sdb` and `/dev/sdc`, but you can substitute different values for your particular nodes.

Although Object Storage supports any file system with extended attributes (xattr), testing and benchmarking indicate the best performance and reliability on XFS. For more information on horizontally scaling your environment, see the Deployment Guide.

This section applies to Red Hat Enterprise Linux 7 and CentOS 7.

Prerequisites

Before you install and configure the Object Storage service on the storage nodes, you must prepare the storage devices.

Note: Perform these steps on each storage node.

1. Install the supporting utility packages:

```
# yum install xfsprogs rsync
```

2. Format the `/dev/sdb` and `/dev/sdc` devices as XFS:

```
# mkfs.xfs /dev/sdb
# mkfs.xfs /dev/sdc
```

3. Create the mount point directory structure:

```
# mkdir -p /srv/node/sdb
# mkdir -p /srv/node/sdc
```

4. Find the UUID of the new partitions:

```
# blkid
```

5. Edit the `/etc/fstab` file and add the following to it:

```
UUID="<UUID-from-output-above>" /srv/node/sdb xfs noatime 0 2
UUID="<UUID-from-output-above>" /srv/node/sdc xfs noatime 0 2
```

6. Mount the devices:

```
# mount /srv/node/sdb
# mount /srv/node/sdc
```

7. Create or edit the `/etc/rsyncd.conf` file to contain the following:

```
uid = swift
gid = swift
log file = /var/log/rsyncd.log
pid file = /var/run/rsyncd.pid
address = MANAGEMENT_INTERFACE_IP_ADDRESS

[account]
max connections = 2
path = /srv/node/
read only = False
lock file = /var/lock/account.lock

[container]
max connections = 2
path = /srv/node/
read only = False
lock file = /var/lock/container.lock

[object]
max connections = 2
path = /srv/node/
read only = False
lock file = /var/lock/object.lock
```

Replace `MANAGEMENT_INTERFACE_IP_ADDRESS` with the IP address of the management network on the storage node.

Note: The `rsync` service requires no authentication, so consider running it on a private network in production environments.

---

7. Start the `rsyncd` service and configure it to start when the system boots:

```
# systemctl enable rsyncd.service
# systemctl start rsyncd.service
```

Install and configure components

Note: Default configuration files vary by distribution. You might need to add these sections and options rather than modifying existing sections and options. Also, an ellipsis (`...`) in the configuration snippets indicates potential default configuration options that you should retain.

Note: Perform these steps on each storage node.

1. Install the packages:

```
# yum install openstack-swift-account openstack-swift-container \
  openstack-swift-object
```

2. Obtain the accounting, container, and object service configuration files from the Object Storage source repository:

```
# curl -o /etc/swift/account-server.conf https://opendev.org/openstack/
↪swift/raw/branch/master/etc/account-server.conf-sample
# curl -o /etc/swift/container-server.conf https://opendev.org/openstack/
↪swift/raw/branch/master/etc/container-server.conf-sample
# curl -o /etc/swift/object-server.conf https://opendev.org/openstack/
↪swift/raw/branch/master/etc/object-server.conf-sample
```

3. Edit the `/etc/swift/account-server.conf` file and complete the following actions:

   - In the `[DEFAULT]` section, configure the bind IP address, bind port, user, configuration directory, and mount point directory:

```
[DEFAULT]
...
bind_ip = MANAGEMENT_INTERFACE_IP_ADDRESS
bind_port = 6202
user = swift
swift_dir = /etc/swift
devices = /srv/node
mount_check = True
```

     Replace `MANAGEMENT_INTERFACE_IP_ADDRESS` with the IP address of the management network on the storage node.

   - In the `[pipeline:main]` section, enable the appropriate modules:

```
[pipeline:main]
pipeline = healthcheck recon account-server
```

     Note: For more information on other modules that enable additional features, see the Deployment Guide.

   - In the `[filter:recon]` section, configure the recon (meters) cache directory:

```
[filter:recon]
use = egg:swift#recon
...
recon_cache_path = /var/cache/swift
```

4. Edit the `/etc/swift/container-server.conf` file and complete the following actions:

   - In the `[DEFAULT]` section, configure the bind IP address, bind port, user, configuration directory, and mount point directory:

```
[DEFAULT]
...
bind_ip = MANAGEMENT_INTERFACE_IP_ADDRESS
bind_port = 6201
user = swift
swift_dir = /etc/swift
devices = /srv/node
mount_check = True
```

     Replace `MANAGEMENT_INTERFACE_IP_ADDRESS` with the IP address of the management network on the storage node.

   - In the `[pipeline:main]` section, enable the appropriate modules:

```
[pipeline:main]
pipeline = healthcheck recon container-server
```

     Note: For more information on other modules that enable additional features, see the Deployment Guide.

   - In the `[filter:recon]` section, configure the recon (meters) cache directory:

```
[filter:recon]
use = egg:swift#recon
...
recon_cache_path = /var/cache/swift
```

5. Edit the `/etc/swift/object-server.conf` file and complete the following actions:

   - In the `[DEFAULT]` section, configure the bind IP address, bind port, user, configuration directory, and mount point directory:

```
[DEFAULT]
...
bind_ip = MANAGEMENT_INTERFACE_IP_ADDRESS
bind_port = 6200
user = swift
swift_dir = /etc/swift
devices = /srv/node
mount_check = True
```

Replace `MANAGEMENT_INTERFACE_IP_ADDRESS` with the IP address of the management network on the storage node.

- In the `[pipeline:main]` section, enable the appropriate modules:

```
[pipeline:main]
pipeline = healthcheck recon object-server
```

Note: For more information on other modules that enable additional features, see the Deployment Guide.

- In the `[filter:recon]` section, configure the recon (meters) cache and lock directories:

```
[filter:recon]
use = egg:swift#recon
...
recon_cache_path = /var/cache/swift
recon_lock_path = /var/lock
```

6. Ensure proper ownership of the mount point directory structure:

```
# chown -R swift:swift /srv/node
```

7. Create the `recon` directory and ensure proper ownership of it:

```
# mkdir -p /var/cache/swift
# chown -R root:swift /var/cache/swift
# chmod -R 775 /var/cache/swift
```

8. Enable necessary access in the firewall

```
# firewall-cmd --permanent --add-port=6200/tcp
# firewall-cmd --permanent --add-port=6201/tcp
# firewall-cmd --permanent --add-port=6202/tcp
```

The rsync service includes its own firewall configuration. Connect from one node to another to ensure that access is allowed.

Install and configure the storage nodes for Ubuntu and Debian

This section describes how to install and configure storage nodes that operate the account, container, and object services. For simplicity, this configuration references two storage nodes, each containing two empty local block storage devices. The instructions use `/dev/sdb` and `/dev/sdc`, but you can substitute different values for your particular nodes.

Although Object Storage supports any file system with extended attributes (xattr), testing and benchmarking indicate the best performance and reliability on XFS. For more information on horizontally scaling your environment, see the Deployment Guide.

This section applies to Ubuntu 14.04 (LTS) and Debian.

Prerequisites

Before you install and configure the Object Storage service on the storage nodes, you must prepare the storage devices.

Note: Perform these steps on each storage node.

1. Install the supporting utility packages:

```
# apt-get install xfsprogs rsync
```

2. Format the /dev/sdb and /dev/sdc devices as XFS:

```
# mkfs.xfs /dev/sdb
# mkfs.xfs /dev/sdc
```

3. Create the mount point directory structure:

```
# mkdir -p /srv/node/sdb
# mkdir -p /srv/node/sdc
```

4. Find the UUID of the new partitions:

```
# blkid
```

5. Edit the /etc/fstab file and add the following to it:

```
UUID="<UUID-from-output-above>" /srv/node/sdb xfs noatime 0 2
UUID="<UUID-from-output-above>" /srv/node/sdc xfs noatime 0 2
```

6. Mount the devices:

```
# mount /srv/node/sdb
# mount /srv/node/sdc
```

7. Create or edit the /etc/rsyncd.conf file to contain the following:

```
uid = swift
gid = swift
log file = /var/log/rsyncd.log
pid file = /var/run/rsyncd.pid
address = MANAGEMENT_INTERFACE_IP_ADDRESS

[account]
max connections = 2
path = /srv/node/
read only = False
lock file = /var/lock/account.lock

[container]
max connections = 2
```

(continues on next page)

(continued from previous page)

```
path = /srv/node/
read only = False
lock file = /var/lock/container.lock

[object]
max connections = 2
path = /srv/node/
read only = False
lock file = /var/lock/object.lock
```

Replace `MANAGEMENT_INTERFACE_IP_ADDRESS` with the IP address of the management network on the storage node.

Note: The `rsync` service requires no authentication, so consider running it on a private network in production environments.

7. Edit the `/etc/default/rsync` file and enable the `rsync` service:

```
RSYNC_ENABLE=true
```

8. Start the `rsync` service:

```
# service rsync start
```

Install and configure components

Note: Default configuration files vary by distribution. You might need to add these sections and options rather than modifying existing sections and options. Also, an ellipsis (`...`) in the configuration snippets indicates potential default configuration options that you should retain.

Note: Perform these steps on each storage node.

1. Install the packages:

```
# apt-get install swift swift-account swift-container swift-object
```

2. Obtain the accounting, container, and object service configuration files from the Object Storage source repository:

```
# curl -o /etc/swift/account-server.conf https://opendev.org/openstack/
↪swift/raw/branch/master/etc/account-server.conf-sample
# curl -o /etc/swift/container-server.conf https://opendev.org/openstack/
↪swift/raw/branch/master/etc/container-server.conf-sample
# curl -o /etc/swift/object-server.conf https://opendev.org/openstack/
↪swift/raw/branch/master/etc/object-server.conf-sample
```

3. Edit the /etc/swift/account-server.conf file and complete the following actions:

   - In the [DEFAULT] section, configure the bind IP address, bind port, user, configuration directory, and mount point directory:

```
[DEFAULT]
...
bind_ip = MANAGEMENT_INTERFACE_IP_ADDRESS
bind_port = 6202
user = swift
swift_dir = /etc/swift
devices = /srv/node
mount_check = True
```

     Replace MANAGEMENT_INTERFACE_IP_ADDRESS with the IP address of the management network on the storage node.

   - In the [pipeline:main] section, enable the appropriate modules:

```
[pipeline:main]
pipeline = healthcheck recon account-server
```

     Note: For more information on other modules that enable additional features, see the Deployment Guide.

   - In the [filter:recon] section, configure the recon (meters) cache directory:

```
[filter:recon]
use = egg:swift#recon
...
recon_cache_path = /var/cache/swift
```

4. Edit the /etc/swift/container-server.conf file and complete the following actions:

   - In the [DEFAULT] section, configure the bind IP address, bind port, user, configuration directory, and mount point directory:

```
[DEFAULT]
...
bind_ip = MANAGEMENT_INTERFACE_IP_ADDRESS
bind_port = 6201
user = swift
swift_dir = /etc/swift
devices = /srv/node
mount_check = True
```

     Replace MANAGEMENT_INTERFACE_IP_ADDRESS with the IP address of the management network on the storage node.

   - In the [pipeline:main] section, enable the appropriate modules:

```
[pipeline:main]
pipeline = healthcheck recon container-server
```

Note: For more information on other modules that enable additional features, see the Deployment Guide.

- In the `[filter:recon]` section, configure the recon (meters) cache directory:

```
[filter:recon]
use = egg:swift#recon
...
recon_cache_path = /var/cache/swift
```

5. Edit the `/etc/swift/object-server.conf` file and complete the following actions:

   - In the `[DEFAULT]` section, configure the bind IP address, bind port, user, configuration directory, and mount point directory:

```
[DEFAULT]
...
bind_ip = MANAGEMENT_INTERFACE_IP_ADDRESS
bind_port = 6200
user = swift
swift_dir = /etc/swift
devices = /srv/node
mount_check = True
```

     Replace `MANAGEMENT_INTERFACE_IP_ADDRESS` with the IP address of the management network on the storage node.

   - In the `[pipeline:main]` section, enable the appropriate modules:

```
[pipeline:main]
pipeline = healthcheck recon object-server
```

Note: For more information on other modules that enable additional features, see the Deployment Guide.

   - In the `[filter:recon]` section, configure the recon (meters) cache and lock directories:

```
[filter:recon]
use = egg:swift#recon
...
recon_cache_path = /var/cache/swift
recon_lock_path = /var/lock
```

6. Ensure proper ownership of the mount point directory structure:

```
# chown -R swift:swift /srv/node
```

7. Create the `recon` directory and ensure proper ownership of it:

```
# mkdir -p /var/cache/swift
# chown -R root:swift /var/cache/swift
# chmod -R 775 /var/cache/swift
```

5.9.5 Create and distribute initial rings

Before starting the Object Storage services, you must create the initial account, container, and object rings. The ring builder creates configuration files that each node uses to determine and deploy the storage architecture. For simplicity, this guide uses one region and two zones with 2^10 (1024) maximum partitions, 3 replicas of each object, and 1 hour minimum time between moving a partition more than once. For Object Storage, a partition indicates a directory on a storage device rather than a conventional partition table. For more information, see the Deployment Guide.

Note: Perform these steps on the controller node.

Create account ring

The account server uses the account ring to maintain lists of containers.

1. Change to the `/etc/swift` directory.
2. Create the base `account.builder` file:

```
# swift-ring-builder account.builder create 10 3 1
```

Note: This command provides no output.

3. Add each storage node to the ring:

```
# swift-ring-builder account.builder \
  add --region 1 --zone 1 --ip STORAGE_NODE_MANAGEMENT_INTERFACE_IP_
↪ADDRESS --port 6202 \
  --device DEVICE_NAME --weight DEVICE_WEIGHT
```

Replace `STORAGE_NODE_MANAGEMENT_INTERFACE_IP_ADDRESS` with the IP address of the management network on the storage node. Replace `DEVICE_NAME` with a storage device name on the same storage node. For example, using the first storage node in *Install and configure the storage nodes* with the `/dev/sdb` storage device and weight of 100:

```
# swift-ring-builder account.builder add \
  --region 1 --zone 1 --ip 10.0.0.51 --port 6202 --device sdb --weight 100
```

Repeat this command for each storage device on each storage node. In the example architecture, use the command in four variations:

```
# swift-ring-builder account.builder add \
  --region 1 --zone 1 --ip 10.0.0.51 --port 6202 --device sdb --weight 100
```

(continues on next page)

(continued from previous page)

```
Device d0r1z1-10.0.0.51:6202R10.0.0.51:6202/sdb_"" with 100.0 weight got␣
↪id 0
# swift-ring-builder account.builder add \
  --region 1 --zone 1 --ip 10.0.0.51 --port 6202 --device sdc --weight 100
Device d1r1z2-10.0.0.51:6202R10.0.0.51:6202/sdc_"" with 100.0 weight got␣
↪id 1
# swift-ring-builder account.builder add \
  --region 1 --zone 2 --ip 10.0.0.52 --port 6202 --device sdb --weight 100
Device d2r1z3-10.0.0.52:6202R10.0.0.52:6202/sdb_"" with 100.0 weight got␣
↪id 2
# swift-ring-builder account.builder add \
  --region 1 --zone 2 --ip 10.0.0.52 --port 6202 --device sdc --weight 100
Device d3r1z4-10.0.0.52:6202R10.0.0.52:6202/sdc_"" with 100.0 weight got␣
↪id 3
```

4. Verify the ring contents:

```
# swift-ring-builder account.builder
account.builder, build version 4
1024 partitions, 3.000000 replicas, 1 regions, 2 zones, 4 devices, 100.00␣
↪balance, 0.00 dispersion
The minimum number of hours before a partition can be reassigned is 1
The overload factor is 0.00% (0.000000)
Devices:    id  region  zone      ip address  port  replication ip ␣
↪replication port      name weight partitions balance meta
             0       1     1       10.0.0.51  6202       10.0.0.51       ␣
↪      6202      sdb  100.00           0 -100.00
             1       1     1       10.0.0.51  6202       10.0.0.51       ␣
↪      6202      sdc  100.00           0 -100.00
             2       1     2       10.0.0.52  6202       10.0.0.52       ␣
↪      6202      sdb  100.00           0 -100.00
             3       1     2       10.0.0.52  6202       10.0.0.52       ␣
↪      6202      sdc  100.00           0 -100.00
```

5. Rebalance the ring:

```
# swift-ring-builder account.builder rebalance
Reassigned 1024 (100.00%) partitions. Balance is now 0.00.  Dispersion is␣
↪now 0.00
```

Create container ring

The container server uses the container ring to maintain lists of objects. However, it does not track object locations.

1. Change to the `/etc/swift` directory.
2. Create the base `container.builder` file:

```
# swift-ring-builder container.builder create 10 3 1
```

Note: This command provides no output.

3. Add each storage node to the ring:

```
# swift-ring-builder container.builder \
  add --region 1 --zone 1 --ip STORAGE_NODE_MANAGEMENT_INTERFACE_IP_
↪ADDRESS --port 6201 \
  --device DEVICE_NAME --weight DEVICE_WEIGHT
```

Replace `STORAGE_NODE_MANAGEMENT_INTERFACE_IP_ADDRESS` with the IP address of the management network on the storage node. Replace `DEVICE_NAME` with a storage device name on the same storage node. For example, using the first storage node in *Install and configure the storage nodes* with the `/dev/sdb` storage device and weight of 100:

```
# swift-ring-builder container.builder add \
  --region 1 --zone 1 --ip 10.0.0.51 --port 6201 --device sdb --weight 100
```

Repeat this command for each storage device on each storage node. In the example architecture, use the command in four variations:

```
# swift-ring-builder container.builder add \
  --region 1 --zone 1 --ip 10.0.0.51 --port 6201 --device sdb --weight 100
Device d0r1z1-10.0.0.51:6201R10.0.0.51:6201/sdb_"" with 100.0 weight got␣
↪id 0
# swift-ring-builder container.builder add \
  --region 1 --zone 1 --ip 10.0.0.51 --port 6201 --device sdc --weight 100
Device d1r1z2-10.0.0.51:6201R10.0.0.51:6201/sdc_"" with 100.0 weight got␣
↪id 1
# swift-ring-builder container.builder add \
  --region 1 --zone 2 --ip 10.0.0.52 --port 6201 --device sdb --weight 100
Device d2r1z3-10.0.0.52:6201R10.0.0.52:6201/sdb_"" with 100.0 weight got␣
↪id 2
# swift-ring-builder container.builder add \
  --region 1 --zone 2 --ip 10.0.0.52 --port 6201 --device sdc --weight 100
Device d3r1z4-10.0.0.52:6201R10.0.0.52:6201/sdc_"" with 100.0 weight got␣
↪id 3
```

4. Verify the ring contents:

```
# swift-ring-builder container.builder
container.builder, build version 4
1024 partitions, 3.000000 replicas, 1 regions, 2 zones, 4 devices, 100.00␣
↪balance, 0.00 dispersion
The minimum number of hours before a partition can be reassigned is 1
The overload factor is 0.00% (0.000000)
Devices:    id  region  zone      ip address  port  replication ip ␣
↪replication port      name weight partitions balance meta
```

(continues on next page)

(continued from previous page)

```
             0       1     1       10.0.0.51  6201       10.0.0.51
↪      6201      sdb  100.00          0 -100.00
             1       1     1       10.0.0.51  6201       10.0.0.51
↪      6201      sdc  100.00          0 -100.00
             2       1     2       10.0.0.52  6201       10.0.0.52
↪      6201      sdb  100.00          0 -100.00
             3       1     2       10.0.0.52  6201       10.0.0.52
↪      6201      sdc  100.00          0 -100.00
```

5. Rebalance the ring:

```
# swift-ring-builder container.builder rebalance
Reassigned 1024 (100.00%) partitions. Balance is now 0.00.  Dispersion is
↪now 0.00
```

Create object ring

The object server uses the object ring to maintain lists of object locations on local devices.

1. Change to the `/etc/swift` directory.
2. Create the base `object.builder` file:

```
# swift-ring-builder object.builder create 10 3 1
```

Note: This command provides no output.

3. Add each storage node to the ring:

```
# swift-ring-builder object.builder \
  add --region 1 --zone 1 --ip STORAGE_NODE_MANAGEMENT_INTERFACE_IP_
↪ADDRESS --port 6200 \
  --device DEVICE_NAME --weight DEVICE_WEIGHT
```

Replace `STORAGE_NODE_MANAGEMENT_INTERFACE_IP_ADDRESS` with the IP address of the management network on the storage node. Replace `DEVICE_NAME` with a storage device name on the same storage node. For example, using the first storage node in *Install and configure the storage nodes* with the `/dev/sdb` storage device and weight of 100:

```
# swift-ring-builder object.builder add \
  --region 1 --zone 1 --ip 10.0.0.51 --port 6200 --device sdb --weight 100
```

Repeat this command for each storage device on each storage node. In the example architecture, use the command in four variations:

```
# swift-ring-builder object.builder add \
  --region 1 --zone 1 --ip 10.0.0.51 --port 6200 --device sdb --weight 100
Device d0r1z1-10.0.0.51:6200R10.0.0.51:6200/sdb_"" with 100.0 weight got
↪id 0
```

(continues on next page)

(continued from previous page)

```
# swift-ring-builder object.builder add \
  --region 1 --zone 1 --ip 10.0.0.51 --port 6200 --device sdc --weight 100
Device d1r1z2-10.0.0.51:6200R10.0.0.51:6200/sdc_"" with 100.0 weight got␣
↪id 1
# swift-ring-builder object.builder add \
  --region 1 --zone 2 --ip 10.0.0.52 --port 6200 --device sdb --weight 100
Device d2r1z3-10.0.0.52:6200R10.0.0.52:6200/sdb_"" with 100.0 weight got␣
↪id 2
# swift-ring-builder object.builder add \
  --region 1 --zone 2 --ip 10.0.0.52 --port 6200 --device sdc --weight 100
Device d3r1z4-10.0.0.52:6200R10.0.0.52:6200/sdc_"" with 100.0 weight got␣
↪id 3
```

4. Verify the ring contents:

```
# swift-ring-builder object.builder
object.builder, build version 4
1024 partitions, 3.000000 replicas, 1 regions, 2 zones, 4 devices, 100.00␣
↪balance, 0.00 dispersion
The minimum number of hours before a partition can be reassigned is 1
The overload factor is 0.00% (0.000000)
Devices:    id  region  zone      ip address  port  replication ip ␣
↪replication port      name weight partitions balance meta
             0       1     1       10.0.0.51  6200       10.0.0.51       ␣
↪      6200      sdb  100.00          0 -100.00
             1       1     1       10.0.0.51  6200       10.0.0.51       ␣
↪      6200      sdc  100.00          0 -100.00
             2       1     2       10.0.0.52  6200       10.0.0.52       ␣
↪      6200      sdb  100.00          0 -100.00
             3       1     2       10.0.0.52  6200       10.0.0.52       ␣
↪      6200      sdc  100.00          0 -100.00
```

5. Rebalance the ring:

```
# swift-ring-builder object.builder rebalance
Reassigned 1024 (100.00%) partitions. Balance is now 0.00.  Dispersion is␣
↪now 0.00
```

Distribute ring configuration files

- Copy the `account.ring.gz`, `container.ring.gz`, and `object.ring.gz` files to the `/etc/swift` directory on each storage node and any additional nodes running the proxy service.

5.9.6 Finalize installation

Finalizing installation varies by distribution.

Finalize installation for openSUSE and SUSE Linux Enterprise

Note: Default configuration files vary by distribution. You might need to add these sections and options rather than modifying existing sections and options. Also, an ellipsis (`...`) in the configuration snippets indicates potential default configuration options that you should retain.

This section applies to openSUSE Leap 42.2 and SUSE Linux Enterprise Server 12 SP2.

1. Edit the `/etc/swift/swift.conf` file and complete the following actions:

   - In the `[swift-hash]` section, configure the hash path prefix and suffix for your environment.

   ```
   [swift-hash]
   ...
   swift_hash_path_suffix = HASH_PATH_SUFFIX
   swift_hash_path_prefix = HASH_PATH_PREFIX
   ```

   Replace HASH_PATH_PREFIX and HASH_PATH_SUFFIX with unique values.

   > **Warning:** Keep these values secret and do not change or lose them.

   - In the `[storage-policy:0]` section, configure the default storage policy:

   ```
   [storage-policy:0]
   ...
   name = Policy-0
   default = yes
   ```

2. Copy the `swift.conf` file to the `/etc/swift` directory on each storage node and any additional nodes running the proxy service.

3. On all nodes, ensure proper ownership of the configuration directory:

   ```
   # chown -R root:swift /etc/swift
   ```

4. On the controller node and any other nodes running the proxy service, start the Object Storage proxy service including its dependencies and configure them to start when the system boots:

   ```
   # systemctl enable openstack-swift-proxy.service memcached.service
   # systemctl start openstack-swift-proxy.service memcached.service
   ```

5. On the storage nodes, start the Object Storage services and configure them to start when the system boots:

```
# systemctl enable openstack-swift-account.service openstack-swift-
↪account-auditor.service \
  openstack-swift-account-reaper.service openstack-swift-account-
↪replicator.service
# systemctl start openstack-swift-account.service openstack-swift-account-
↪auditor.service \
  openstack-swift-account-reaper.service openstack-swift-account-
↪replicator.service
# systemctl enable openstack-swift-container.service openstack-swift-
↪container-auditor.service \
  openstack-swift-container-replicator.service openstack-swift-container-
↪updater.service
# systemctl start openstack-swift-container.service openstack-swift-
↪container-auditor.service \
  openstack-swift-container-replicator.service openstack-swift-container-
↪updater.service
# systemctl enable openstack-swift-object.service openstack-swift-object-
↪auditor.service \
  openstack-swift-object-replicator.service openstack-swift-object-
↪updater.service
# systemctl start openstack-swift-object.service openstack-swift-object-
↪auditor.service \
  openstack-swift-object-replicator.service openstack-swift-object-
↪updater.service
```

Finalize installation for Red Hat Enterprise Linux and CentOS

Note: Default configuration files vary by distribution. You might need to add these sections and options rather than modifying existing sections and options. Also, an ellipsis (`...`) in the configuration snippets indicates potential default configuration options that you should retain.

This section applies to Red Hat Enterprise Linux 7 and CentOS 7.

1. Obtain the `/etc/swift/swift.conf` file from the Object Storage source repository:

```
# curl -o /etc/swift/swift.conf \
  https://opendev.org/openstack/swift/raw/branch/master/etc/swift.conf-
↪sample
```

2. Edit the `/etc/swift/swift.conf` file and complete the following actions:

   - In the `[swift-hash]` section, configure the hash path prefix and suffix for your environment.

```
[swift-hash]
...
swift_hash_path_suffix = HASH_PATH_SUFFIX
swift_hash_path_prefix = HASH_PATH_PREFIX
```

Replace HASH_PATH_PREFIX and HASH_PATH_SUFFIX with unique values.

> **Warning:** Keep these values secret and do not change or lose them.

- In the `[storage-policy:0]` section, configure the default storage policy:

```
[storage-policy:0]
...
name = Policy-0
default = yes
```

3. Copy the `swift.conf` file to the `/etc/swift` directory on each storage node and any additional nodes running the proxy service.
4. On all nodes, ensure proper ownership of the configuration directory:

```
# chown -R root:swift /etc/swift
```

5. On the controller node and any other nodes running the proxy service, start the Object Storage proxy service including its dependencies and configure them to start when the system boots:

```
# systemctl enable openstack-swift-proxy.service memcached.service
# systemctl start openstack-swift-proxy.service memcached.service
```

6. On the storage nodes, start the Object Storage services and configure them to start when the system boots:

```
# systemctl enable openstack-swift-account.service openstack-swift-
↪account-auditor.service \
  openstack-swift-account-reaper.service openstack-swift-account-
↪replicator.service
# systemctl start openstack-swift-account.service openstack-swift-account-
↪auditor.service \
  openstack-swift-account-reaper.service openstack-swift-account-
↪replicator.service
# systemctl enable openstack-swift-container.service \
  openstack-swift-container-auditor.service openstack-swift-container-
↪replicator.service \
  openstack-swift-container-updater.service
# systemctl start openstack-swift-container.service \
  openstack-swift-container-auditor.service openstack-swift-container-
↪replicator.service \
  openstack-swift-container-updater.service
# systemctl enable openstack-swift-object.service openstack-swift-object-
↪auditor.service \
  openstack-swift-object-replicator.service openstack-swift-object-
↪updater.service
# systemctl start openstack-swift-object.service openstack-swift-object-
↪auditor.service \
  openstack-swift-object-replicator.service openstack-swift-object-
↪updater.service
```

Finalize installation for Ubuntu and Debian

Note: Default configuration files vary by distribution. You might need to add these sections and options rather than modifying existing sections and options. Also, an ellipsis (`...`) in the configuration snippets indicates potential default configuration options that you should retain.

This section applies to Ubuntu 14.04 (LTS) and Debian.

1. Obtain the `/etc/swift/swift.conf` file from the Object Storage source repository:

```
# curl -o /etc/swift/swift.conf \
  https://opendev.org/openstack/swift/raw/branch/master/etc/swift.conf-
↪sample
```

2. Edit the `/etc/swift/swift.conf` file and complete the following actions:

   - In the `[swift-hash]` section, configure the hash path prefix and suffix for your environment.

```
[swift-hash]
...
swift_hash_path_suffix = HASH_PATH_SUFFIX
swift_hash_path_prefix = HASH_PATH_PREFIX
```

     Replace HASH_PATH_PREFIX and HASH_PATH_SUFFIX with unique values.

     Warning: Keep these values secret and do not change or lose them.

   - In the `[storage-policy:0]` section, configure the default storage policy:

```
[storage-policy:0]
...
name = Policy-0
default = yes
```

3. Copy the `swift.conf` file to the `/etc/swift` directory on each storage node and any additional nodes running the proxy service.

4. On all nodes, ensure proper ownership of the configuration directory:

```
# chown -R root:swift /etc/swift
```

5. On the controller node and any other nodes running the proxy service, restart the Object Storage proxy service including its dependencies:

```
# service memcached restart
# service swift-proxy restart
```

6. On the storage nodes, start the Object Storage services:

```
# swift-init all start
```

Note: The storage node runs many Object Storage services and the **swift-init** command makes them easier to manage. You can ignore errors from services not running on the storage node.

5.9.7 Verify operation

Verify operation of the Object Storage service.

Note: Perform these steps on the controller node.

Warning: If you are using Red Hat Enterprise Linux 7 or CentOS 7 and one or more of these steps do not work, check the `/var/log/audit/audit.log` file for SELinux messages indicating denial of actions for the `swift` processes. If present, change the security context of the `/srv/node` directory to the lowest security level (s0) for the `swift_data_t` type, `object_r` role and the `system_u` user:

```
# chcon -R system_u:object_r:swift_data_t:s0 /srv/node
```

1. Source the `demo` credentials:

```
$ . demo-openrc
```

2. Show the service status:

```
$ swift stat
                        Account: AUTH_ed0b60bf607743088218b0a533d5943f
                     Containers: 0
                        Objects: 0
                          Bytes: 0
   X-Account-Project-Domain-Id: default
                    X-Timestamp: 1444143887.71539
                     X-Trans-Id: tx1396aeaf17254e94beb34-0056143bde
          X-Openstack-Request-Id: tx1396aeaf17254e94beb34-0056143bde
                  Content-Type: text/plain; charset=utf-8
                 Accept-Ranges: bytes
```

3. Create `container1` container:

```
$ openstack container create container1
+---------------------------------------+------------+------------------------
↪----------------+
| account                               | container  | x-trans-id             ␣
↪               |
+---------------------------------------+------------+------------------------
↪----------------+
| AUTH_ed0b60bf607743088218b0a533d5943f | container1 |␣
↪tx8c4034dc306c44dd8cd68-0056f00a4a |
+---------------------------------------+------------+------------------------
↪----------------+
```

4. Upload a test file to the `container1` container:

```
$ openstack object create container1 FILE
+--------+------------+----------------------------------+
| object | container  | etag                             |
+--------+------------+----------------------------------+
| FILE   | container1 | ee1eca47dc88f4879d8a229cc70a07c6 |
+--------+------------+----------------------------------+
```

   Replace `FILE` with the name of a local file to upload to the `container1` container.

5. List files in the `container1` container:

```
$ openstack object list container1
+------+
| Name |
+------+
| FILE |
+------+
```

6. Download a test file from the `container1` container:

```
$ openstack object save container1 FILE
```

   Replace `FILE` with the name of the file uploaded to the `container1` container.

   Note: This command provides no output.

5.9.8 Next steps

Your OpenStack environment now includes Object Storage.

To add more services, see the additional documentation on installing OpenStack .

The Object Storage services (swift) work together to provide object storage and retrieval through a REST API.

This chapter assumes a working setup of OpenStack following the OpenStack Installation Tutorial.

Your environment must at least include the Identity service (keystone) prior to deploying Object Storage.

5.10 Configuration Documentation

5.10.1 Common configuration

This document describes the configuration options common to all swift servers. Documentation for other swift configuration options can be found at *Configuration Documentation*.

An example of common configuration file can be found at etc/swift.conf-sample

The following configuration options are available:

Option	Default	Description
max_header_size	8192	max_header_size is the max number of bytes in the utf8 encoding of each header. Using 8192 as default because eventlet use 8192 as max size of header line. This value may need to be increased when using identity v3 API tokens including more than 7 catalog entries. See also include_service_catalog in proxy-server.conf-sample (documented in overview_auth.rst).
extra_header_count	0	By default the maximum number of allowed headers depends on the number of max allowed metadata settings plus a default value of 32 for regular http headers. If for some reason this is not enough (custom middleware for example) it can be increased with the extra_header_count constraint.
auto_create_account_prefix	.	Prefix used when automatically creating accounts.

5.10.2 Proxy Server Configuration

This document describes the configuration options available for the proxy server. Some proxy server options may be configured on a *per-policy* basis. Additional documentation for proxy-server middleware can be found at *Middleware* and *The Auth System*.

Documentation for other swift configuration options can be found at *Configuration Documentation*.

An example Proxy Server configuration can be found at etc/proxy-server.conf-sample in the source code repository.

The following configuration sections are available:

- *[DEFAULT]*
- *[proxy-server]*

[DEFAULT]

Option	Default	Description
bind_ip	0.0.0.0	IP Address for server to bind to
bind_port	80	Port for server to bind to
keep_idle	600	Value to set for socket TCP_KEEPIDLE
bind_timeout	30	Seconds to attempt bind before giving up
backlog	4096	Maximum number of allowed pending connections
swift_dir	/etc/swift	Swift configuration directory
workers	auto	Override the number of pre-forked workers that will a
max_clients	1024	Maximum number of clients one worker can process
user	swift	User to run as
cert_file		Path to the ssl .crt. This should be enabled for testing
key_file		Path to the ssl .key. This should be enabled for testing
cors_allow_origin		List of origin hosts that are allowed for CORS reques
strict_cors_mode	True	If True (default) then CORS requests are only allowed
cors_expose_headers		This is a list of headers that are included in the heade
client_timeout	60	
trans_id_suffix		This optional suffix (default is empty) that would be a

log_name	swift	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_headers	False	
log_address	/dev/log	Logging directory
log_max_line_length	0	Caps the length of log lines to the value given; no lim
log_custom_handlers	None	Comma separated list of functions to call to setup cus
log_udp_host		Override log_address
log_udp_port	514	UDP log port
log_statsd_host	None	Enables StatsD logging; IPv4/IPv6 address or a hostn
log_statsd_port	8125	
log_statsd_default_sample_rate	1.0	
log_statsd_sample_rate_factor	1.0	
log_statsd_metric_prefix		
eventlet_debug	false	If true, turn on debug logging for eventlet
expose_info	true	Enables exposing configuration settings via HTTP GI
admin_key		Key to use for admin calls that are HMAC signed. De
disallowed_sections	swift.valid_api_versions	Allows the ability to withhold sections from showing
expiring_objects_container_divisor	86400	
expiring_objects_account_name	expiring_objects	
nice_priority	None	Scheduling priority of server processes. Niceness val
ionice_class	None	I/O scheduling class of server processes. I/O niceness
ionice_priority	None	I/O scheduling priority of server processes. I/O nicen

[proxy-server]

Option	Default	Description
use		Entry point for paste.
set log_name	proxy-server	Label used when log
set log_facility	LOG_LOCAL0	Syslog log facility
set log_level	INFO	Log level
set log_headers	True	If True, log headers i
set log_handoffs	True	If True, the proxy wi
recheck_account_existence	60	Cache timeout in sec
recheck_container_existence	60	Cache timeout in sec
object_chunk_size	65536	Chunk size to read fr
client_chunk_size	65536	Chunk size to read fr
memcache_servers	127.0.0.1:11211	Comma separated lis
memcache_max_connections	2	Max number of conn
node_timeout	10	Request timeout to ex
recoverable_node_timeout	node_timeout	Request timeout to ex
client_timeout	60	Timeout to read one
conn_timeout	0.5	Connection timeout t
error_suppression_interval	60	Time in seconds that
error_suppression_limit	10	Error count to consid
allow_account_management	false	Whether account PU

account_autocreate	false	If set to true authoriz
max_containers_per_account	0	If set to a positive va
max_containers_whitelist		This is a comma sepa
rate_limit_after_segment	10	Rate limit the downlo
rate_limit_segments_per_sec	1	Rate limit large objec
request_node_count	2 * replicas	Set to the number of
swift_owner_headers	<see the sample conf file for the list of default headers>	These are the header
sorting_method	shuffle	Storage nodes can be
timing_expiry	300	If the timing sorting_
concurrent_gets	off	Use replica count nu
concurrency_timeout	conn_timeout	This parameter contr
nice_priority	None	Scheduling priority c
ionice_class	None	I/O scheduling class
ionice_priority	None	I/O scheduling priori
read_affinity	None	Specifies which back
write_affinity	None	Specifies which back
write_affinity_node_count	2 * replicas	The number of local
write_affinity_handoff_delete_count	auto	The number of local

5.10.3 Account Server Configuration

This document describes the configuration options available for the account server. Documentation for other swift configuration options can be found at *Configuration Documentation*.

An example Account Server configuration can be found at etc/account-server.conf-sample in the source code repository.

The following configuration sections are available:

- *[DEFAULT]*
- *[account-server]*
- *[account-replicator]*
- *[account-auditor]*
- *[account-reaper]*

[DEFAULT]

Option	Default	Description
swift_dir	/etc/swift	Swift configuration directory
devices	/srv/node	Parent directory or where devices are mounted
mount_check	true	Whether or not check if the devices are mounted to prevent accide
bind_ip	0.0.0.0	IP Address for server to bind to
bind_port	6202	Port for server to bind to
keep_idle	600	Value to set for socket TCP_KEEPIDLE
bind_timeout	30	Seconds to attempt bind before giving up
backlog	4096	Maximum number of allowed pending connections

workers	auto	Override the number of pre-forked workers that will accept conne
max_clients	1024	Maximum number of clients one worker can process simultaneou
user	swift	User to run as
db_preallocation	off	If you dont mind the extra disk space usage in overhead, you can t
disable_fallocate	false	Disable fast fail fallocate checks if the underlying filesystem does
log_name	swift	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
log_max_line_length	0	Caps the length of log lines to the value given; no limit if set to 0,
log_custom_handlers	None	Comma-separated list of functions to call to setup custom log han
log_udp_host		Override log_address
log_udp_port	514	UDP log port
log_statsd_host	None	Enables StatsD logging; IPv4/IPv6 address or a hostname. If a ho
log_statsd_port	8125	
log_statsd_default_sample_rate	1.0	
log_statsd_sample_rate_factor	1.0	
log_statsd_metric_prefix		
eventlet_debug	false	If true, turn on debug logging for eventlet
fallocate_reserve	1%	You can set fallocate_reserve to the number of bytes or percentag
nice_priority	None	Scheduling priority of server processes. Niceness values range fr
ionice_class	None	I/O scheduling class of server processes. I/O niceness class value
ionice_priority	None	I/O scheduling priority of server processes. I/O niceness priority

[account-server]

Option	Default	Description
use		Entry point for paste.deploy for the account server. For most cases, this should be `egg:swift#account`.
set log_name	account-server	Label used when logging
set log_facility	LOG_LOCAL0	Syslog log facility
set log_level	INFO	Logging level
set log_requests	True	Whether or not to log each request
set log_address	/dev/log	Logging directory
replication_server		Configure parameter for creating specific server. To handle all verbs, including replication verbs, do not specify replication_server (this is the default). To only handle replication, set to a True value (e.g. True or 1). To handle only non-replication verbs, set to False. Unless you have a separate replication network, you should not specify any value for replication_server.
nice_priority	None	Scheduling priority of server processes. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process). The default does not modify priority.
ionice_class	None	I/O scheduling class of server processes. I/O niceness class values are IOPRIO_CLASS_RT (realtime), IOPRIO_CLASS_BE (best-effort), and IOPRIO_CLASS_IDLE (idle). The default does not modify class and priority. Linux supports io scheduling priorities and classes since 2.6.13 with the CFQ io scheduler. Work only with ionice_priority.
ionice_priority	None	I/O scheduling priority of server processes. I/O niceness priority is a number which goes from 0 to 7. The higher the value, the lower the I/O priority of the process. Work only with ionice_class. Ignored if IOPRIO_CLASS_IDLE is set.

[account-replicator]

Option	Default	Description
log_name	account-replicator	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
per_diff	1000	Maximum number of database rows that will be syncd in a single HTTP replication request. Databases with less than or equal to this number of differing rows will always be syncd using an HTTP replication request rather than using rsync.
max_diffs	100	Maximum number of HTTP replication requests attempted on each replication pass for any one container. This caps how long the replicator will spend trying to sync a given database per pass so the other databases dont get starved.
concurrency	8	Number of replication workers to spawn
interval	30	Time in seconds to wait between replication passes
databases_per_second	50	Maximum databases to process per second. Should be tuned according to individual system specs. 0 is unlimited.
node_timeout	10	Request timeout to external services
conn_timeout	0.5	Connection timeout to external services
reclaim_age	604800	Time elapsed in seconds before an account can be reclaimed
rsync_module	{replication_ip}::account	Format of the rsync module where the replicator will send data. The configuration value can include some variables that will be extracted from the ring. Variables must follow the format {NAME} where NAME is one of: ip, port, replication_ip, replication_port, region, zone, device, meta. See etc/rsyncd.conf-sample for some examples.
rsync_compress	no	Allow rsync to compress data which is transmitted to destination node during sync. However, this is applicable only when destination node is in a different region than the local one. NOTE: Objects that are already compressed (for example: .tar.gz, mp3) might slow down the syncing process.
recon_cache_path	/var/cache/swift	Path to recon cache
nice_priority	None	Scheduling priority of server processes. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process). The default does not modify priority.
ionice_class	None	I/O scheduling class of server processes. I/O niceness class values are IOPRIO_CLASS_RT (realtime), IOPRIO_CLASS_BE (best-effort), and IOPRIO_CLASS_IDLE (idle). The default does not modify class and priority. Linux supports io scheduling priorities and classes since 2.6.13 with the CFQ io scheduler. Work only with ionice_priority.
ionice_priority	None	I/O scheduling priority of server processes. I/O niceness priority is a number which goes from 0 to 7. The higher the value, the lower the I/O priority of the process. Work only with ionice_class. Ignored if IOPRIO_CLASS_IDLE is set.
handoffs_only	no	When handoffs_only mode is enabled the replicator will *only* replicate from handoff nodes to primary nodes and will not sync primary nodes with other primary nodes.
handoff_delete	auto	the number of replicas which are ensured in swift. If the number less than the number of replicas is set, account-replicator could delete local handoffs even if all replicas are not ensured in the cluster. The replicator would remove local handoff account database after syncing when the number of successful responses is greater than or equal to this number. By default handoff partitions will be removed when it has successfully replicated to all

[account-auditor]

Option	Default	Description
log_name	account-auditor	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
interval	1800	Minimum time for a pass to take
accounts_per_second	200	Maximum accounts audited per second. Should be tuned according to individual system specs. 0 is unlimited.
recon_cache_path	/var/cache/swift	Path to recon cache
nice_priority	None	Scheduling priority of server processes. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process). The default does not modify priority.
ionice_class	None	I/O scheduling class of server processes. I/O niceness class values are IOPRIO_CLASS_RT (realtime), IOPRIO_CLASS_BE (best-effort), and IOPRIO_CLASS_IDLE (idle). The default does not modify class and priority. Linux supports io scheduling priorities and classes since 2.6.13 with the CFQ io scheduler. Work only with ionice_priority.
ionice_priority	None	I/O scheduling priority of server processes. I/O niceness priority is a number which goes from 0 to 7. The higher the value, the lower the I/O priority of the process. Work only with ionice_class. Ignored if IOPRIO_CLASS_IDLE is set.

[account-reaper]

Option	Default	Description
log_name	account-reaper	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
concurrency	25	Number of replication workers to spawn
interval	3600	Minimum time for a pass to take
node_timeout	10	Request timeout to external services
conn_timeout	0.5	Connection timeout to external services
delay_reaping	0	Normally, the reaper begins deleting account information for deleted accounts immediately; you can set this to delay its work however. The value is in seconds, 2592000 = 30 days, for example. The sum of this value and the container-updater `interval` should be less than the account-replicator `reclaim_age`. This ensures that once the account-reaper has deleted a container there is sufficient time for the container-updater to report to the account before the account DB is removed.
reap_warn_after	2892000	If the account fails to be reaped due to a persistent error, the account reaper will log a message such as: Account <name> has not been reaped since <date> You can search logs for this message if space is not being reclaimed after you delete account(s). This is in addition to any time requested by delay_reaping.
nice_priority	None	Scheduling priority of server processes. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process). The default does not modify priority.
ionice_class	None	I/O scheduling class of server processes. I/O niceness class values are IOPRIO_CLASS_RT (realtime), IOPRIO_CLASS_BE (best-effort), and IOPRIO_CLASS_IDLE (idle). The default does not modify class and priority. Linux supports io scheduling priorities and classes since 2.6.13 with the CFQ io scheduler. Work only with ionice_priority.
ionice_priority	None	I/O scheduling priority of server processes. I/O niceness priority is a number which goes from 0 to 7. The higher the value, the lower the I/O priority of the process. Work only with ionice_class. Ignored if IOPRIO_CLASS_IDLE is set.

5.10.4 Container Server Configuration

This document describes the configuration options available for the container server. Documentation for other swift configuration options can be found at *Configuration Documentation*.

An example Container Server configuration can be found at etc/container-server.conf-sample in the source code repository.

The following configuration sections are available:

- *[DEFAULT]*
- *[container-server]*

- *[container-replicator]*
- *[container-sharder]*
- *[container-updater]*
- *[container-auditor]*

[DEFAULT]

Option	Default	Description
swift_dir	/etc/swift	Swift configuration directory
devices	/srv/node	Parent directory of where devices are mounted
mount_check	true	Whether or not check if the devices are mounted to prevent accide
bind_ip	0.0.0.0	IP Address for server to bind to
bind_port	6201	Port for server to bind to
keep_idle	600	Value to set for socket TCP_KEEPIDLE
bind_timeout	30	Seconds to attempt bind before giving up
backlog	4096	Maximum number of allowed pending connections
workers	auto	Override the number of pre-forked workers that will accept conne
max_clients	1024	Maximum number of clients one worker can process simultaneou
user	swift	User to run as
disable_fallocate	false	Disable fast fail fallocate checks if the underlying filesystem does
log_name	swift	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
log_max_line_length	0	Caps the length of log lines to the value given; no limit if set to 0,
log_custom_handlers	None	Comma-separated list of functions to call to setup custom log han
log_udp_host		Override log_address
log_udp_port	514	UDP log port
log_statsd_host	None	Enables StatsD logging; IPv4/IPv6 address or a hostname. If a ho
log_statsd_port	8125	
log_statsd_default_sample_rate	1.0	
log_statsd_sample_rate_factor	1.0	
log_statsd_metric_prefix		
eventlet_debug	false	If true, turn on debug logging for eventlet
fallocate_reserve	1%	You can set fallocate_reserve to the number of bytes or percentag
db_preallocation	off	If you dont mind the extra disk space usage in overhead, you can t
nice_priority	None	Scheduling priority of server processes. Niceness values range fr
ionice_class	None	I/O scheduling class of server processes. I/O niceness class value
ionice_priority	None	I/O scheduling priority of server processes. I/O niceness priority

[container-server]

Option	Default	Description
use		paste.deploy entry point for the container server. For most cases, this should be `egg:swift#container`.
set log_name	container-server	Label used when logging
set log_facility	LOG_LOCAL0	Syslog log facility
set log_level	INFO	Logging level
set log_requests	True	Whether or not to log each request
set log_address	/dev/log	Logging directory
node_timeout	3	Request timeout to external services
conn_timeout	0.5	Connection timeout to external services
allow_versions	false	Enable/Disable object versioning feature
replication_server		Configure parameter for creating specific server. To handle all verbs, including replication verbs, do not specify replication_server (this is the default). To only handle replication, set to a True value (e.g. True or 1). To handle only non-replication verbs, set to False. Unless you have a separate replication network, you should not specify any value for replication_server.
nice_priority	None	Scheduling priority of server processes. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process). The default does not modify priority.
ionice_class	None	I/O scheduling class of server processes. I/O niceness class values are IOPRIO_CLASS_RT (realtime), IOPRIO_CLASS_BE (best-effort), and IOPRIO_CLASS_IDLE (idle). The default does not modify class and priority. Linux supports io scheduling priorities and classes since 2.6.13 with the CFQ io scheduler. Work only with ionice_priority.
ionice_priority	None	I/O scheduling priority of server processes. I/O niceness priority is a number which goes from 0 to 7. The higher the value, the lower the I/O priority of the process. Work only with ionice_class. Ignored if IOPRIO_CLASS_IDLE is set.

[container-replicator]

Option	Default	Description
log_name	container-replicator	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
per_diff	1000	Maximum number of database rows that will be syncd in a single HTTP replication request. Databases with less than or equal to this number of differing rows will always be syncd using an HTTP replication request rather than using rsync.
max_diffs	100	Maximum number of HTTP replication requests attempted on each replication pass for any one container. This caps how long the replicator will spend trying to sync a given database per pass so the other databases dont get starved.
concurrency	8	Number of replication workers to spawn
interval	30	Time in seconds to wait between replication passes
databases_per_second	50	Maximum databases to process per second. Should be tuned according to individual system specs. 0 is unlimited.
node_timeout	10	Request timeout to external services
conn_timeout	0.5	Connection timeout to external services
reclaim_age	604800	Time elapsed in seconds before a container can be reclaimed
rsync_module	{replication_ip}::container	Format of the rsync module where the replicator will send data. The configuration value can include some variables that will be extracted from the ring. Variables must follow the format {NAME} where NAME is one of: ip, port, replication_ip, replication_port, region, zone, device, meta. See etc/rsyncd.conf-sample for some examples.
rsync_compress	no	Allow rsync to compress data which is transmitted to destination node during sync. However, this is applicable only when destination node is in a different region than the local one. NOTE: Objects that are already compressed (for example: .tar.gz, mp3) might slow down the syncing process.
recon_cache_path	/var/cache/swift	Path to recon cache
nice_priority	None	Scheduling priority of server processes. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process). The default does not modify priority.
ionice_class	None	I/O scheduling class of server processes. I/O niceness class values are IOPRIO_CLASS_RT (realtime), IOPRIO_CLASS_BE (best-effort), and IOPRIO_CLASS_IDLE (idle). The default does not modify class and priority. Linux supports io scheduling priorities and classes since 2.6.13 with the CFQ io scheduler. Work only with ionice_priority.
ionice_priority	None	I/O scheduling priority of server processes. I/O niceness priority is a number which goes from 0 to 7. The higher the value, the lower the I/O priority of the process. Work only with ionice_class. Ignored if IOPRIO_CLASS_IDLE is set.
handoffs_only	no	When handoffs_only mode is enabled the replicator will *only* replicate from handoff nodes to primary nodes and will not sync primary nodes with other primary nodes.
handoff_delete	auto	the number of replicas which are ensured in swift. If the number less than the number of replicas is set, container-replicator could delete local handoffs even if all replicas are not ensured in the cluster. The replicator would remove local handoff container database after syncing when the number of successful responses is greater than or equal to this number. By default handoff partitions will be removed when it has successfully

[container-sharder]

The container-sharder re-uses features of the container-replicator and inherits the following configuration options defined for the *[container-replicator]*:

- interval
- databases_per_second
- per_diff
- max_diffs
- concurrency
- node_timeout
- conn_timeout
- reclaim_age
- rsync_compress
- rsync_module
- recon_cache_path

Some config options in this section may also be used by the *swift-manage-shard-ranges CLI tool*.

Option	Default	Description
log_name	container-sharder	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
auto_shard	false	If the auto_shard option is true then the sharder will automatically select containers to shard, scan for shard ranges, and select shards to shrink. Warning: auto-sharding is still under development and should not be used in production; do not set this option to true in a production cluster.
shard_container_threshold	1000000	This defines the object count at which a container with container-sharding enabled will start to shard. This also indirectly determines the the defaults for rows_per_shard, shrink_threshold and expansion_limit.
rows_per_shard	500000	This defines the initial nominal size of shard containers. The default is shard_container_threshold // 2.
minimum_shard_size	100000	Minimum size of the final shard range. If this is greater than one then the final shard range may be extended to more than rows_per_shard in order to avoid a further shard range with less than minimum_shard_size rows. The default value is rows_per_shard // 5.
shrink_threshold		This defines the object count below which a donor shard container will be considered for shrinking into another acceptor shard container. The default is determined by shard_shrink_point. If set, shrink_threshold will take precedence over shard_shrink_point.
shard_shrink_point	10	Deprecated: shrink_threshold is recommended and if set will take precedence over shard_shrink_point. This defines the object count below which a donor shard container will be considered for shrinking into another acceptor shard container. shard_shrink_point is a percentage of shard_container_threshold e.g. the default value of 10 means 10% of the shard_container_threshold.
expansion_limit		This defines the maximum allowed size of an acceptor shard container after having a donor merged into it. The default is determined by shard_shrink_merge_point. If set, expansion_limit will take precedence over shard_shrink_merge_point.
shard_shrink_merge_point	75	Deprecated: expansion_limit is recommended and if set will take precedence over shard_shrink_merge_point. This defines the maximum allowed size of an acceptor shard container after having a donor merged into it. Shard_shrink_merge_point is a percentage of shard_container_threshold. e.g. the default value of 75 means that the projected sum of a donor object count and acceptor count must be less than 75% of shard_container_threshold for the donor to be allowed to merge into the acceptor. For example, if shard_container_threshold is 1 million, shard_shrink_point is 10, and shard_shrink_merge_point is 75 then a shard will be considered for shrinking if it has less than or equal to 100 thousand objects but will only merge into an acceptor if the combined object count would be less than or equal to 750 thousand objects.
shard_scanner_batch_size	10	When auto-sharding is enabled this defines the maximum number of shard ranges that will be found each time the sharder daemon visits a sharding container. If necessary the sharder daemon will continue to search for more shard ranges each time it visits the container.
cleave_batch_size	2	Defines the number of shard ranges that will be cleaved each time the sharder daemon visits a sharding container.
cleave_row_batch_size	10000	Defines the size of batches of object rows read from a sharding container and merged to a shard container during cleaving.
shard_replication_quorum	auto	Defines the number of successfully replicated shard dbs required when cleaving a previously uncleaved shard range before the sharder will progress to the next shard range. The value should be less than or equal to the container ring replica count. The default of auto causes the container ring quorum value to be used. This option only applies to the container-sharder replication and does not affect the number of shard container replicas that will eventually be replicated by the container replicator.

[container-updater]

Option	Default	Description
log_name	container-updater	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
interval	300	Minimum time for a pass to take
concurrency	4	Number of updater workers to spawn
node_timeout	3	Request timeout to external services
conn_timeout	0.5	Connection timeout to external services
containers_per_second	50	Maximum containers updated per second. Should be tuned according to individual system specs. 0 is unlimited.
slowdown	0.01	Time in seconds to wait between containers. Deprecated in favor of containers_per_second.
account_suppression_time	60	Seconds to suppress updating an account that has generated an error (timeout, not yet found, etc.)
recon_cache_path	/var/cache/swift	Path to recon cache
nice_priority	None	Scheduling priority of server processes. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process). The default does not modify priority.
ionice_class	None	I/O scheduling class of server processes. I/O niceness class values are IOPRIO_CLASS_RT (realtime), IOPRIO_CLASS_BE (best-effort), and IOPRIO_CLASS_IDLE (idle). The default does not modify class and priority. Linux supports io scheduling priorities and classes since 2.6.13 with the CFQ io scheduler. Work only with ionice_priority.
ionice_priority	None	I/O scheduling priority of server processes. I/O niceness priority is a number which goes from 0 to 7. The higher the value, the lower the I/O priority of the process. Work only with ionice_class. Ignored if IOPRIO_CLASS_IDLE is set.

[container-auditor]

Option	Default	Description
log_name	container-auditor	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
interval	1800	Minimum time for a pass to take
containers_per_second	200	Maximum containers audited per second. Should be tuned according to individual system specs. 0 is unlimited.
recon_cache_path	/var/cache/swift	Path to recon cache
nice_priority	None	Scheduling priority of server processes. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process). The default does not modify priority.
ionice_class	None	I/O scheduling class of server processes. I/O niceness class values are IOPRIO_CLASS_RT (realtime), IOPRIO_CLASS_BE (best-effort), and IOPRIO_CLASS_IDLE (idle). The default does not modify class and priority. Linux supports io scheduling priorities and classes since 2.6.13 with the CFQ io scheduler. Work only with ionice_priority.
ionice_priority	None	I/O scheduling priority of server processes. I/O niceness priority is a number which goes from 0 to 7. The higher the value, the lower the I/O priority of the process. Work only with ionice_class. Ignored if IOPRIO_CLASS_IDLE is set.

5.10.5 Object Server Configuration

This document describes the configuration options available for the object server. Documentation for other swift configuration options can be found at *Configuration Documentation*.

An Example Object Server configuration can be found at etc/object-server.conf-sample in the source code repository.

The following configuration sections are available:

- *[DEFAULT]*
- *[object-server]*
- *[object-replicator]*
- *[object-reconstructor]*
- *[object-updater]*
- *[object-auditor]*
- *[object-expirer]*

[DEFAULT]

Option	Default	Description
swift_dir	/etc/swift	Swift configuration directory
devices	/srv/node	Parent directory of where devices are mounted
mount_check	true	Whether or not check if the devices are mounted to prevent accide
bind_ip	0.0.0.0	IP Address for server to bind to
bind_port	6200	Port for server to bind to
keep_idle	600	Value to set for socket TCP_KEEPIDLE
bind_timeout	30	Seconds to attempt bind before giving up
backlog	4096	Maximum number of allowed pending connections
workers	auto	Override the number of pre-forked workers that will accept conne
servers_per_port	0	If each disk in each storage policy ring has unique port numbers f
max_clients	1024	Maximum number of clients one worker can process simultaneou
disable_fallocate	false	Disable fast fail fallocate checks if the underlying filesystem does
log_name	swift	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
log_max_line_length	0	Caps the length of log lines to the value given; no limit if set to 0,
log_custom_handlers	None	Comma-separated list of functions to call to setup custom log han
log_udp_host		Override log_address
log_udp_port	514	UDP log port
log_statsd_host	None	Enables StatsD logging; IPv4/IPv6 address or a hostname. If a ho
log_statsd_port	8125	
log_statsd_default_sample_rate	1.0	
log_statsd_sample_rate_factor	1.0	
log_statsd_metric_prefix		
eventlet_debug	false	If true, turn on debug logging for eventlet
fallocate_reserve	1%	You can set fallocate_reserve to the number of bytes or percentag
conn_timeout	0.5	Time to wait while attempting to connect to another backend node
node_timeout	3	Time to wait while sending each chunk of data to another backend
client_timeout	60	Time to wait while receiving each chunk of data from a client or a
network_chunk_size	65536	Size of chunks to read/write over the network
disk_chunk_size	65536	Size of chunks to read/write to disk
container_update_timeout	1	Time to wait while sending a container update on object update.
reclaim_age	604800	Time elapsed in seconds before the tombstone file representing a
commit_window	60	Non-durable data files may also get reclaimed if they are older tha
nice_priority	None	Scheduling priority of server processes. Niceness values range fro
ionice_class	None	I/O scheduling class of server processes. I/O niceness class value
ionice_priority	None	I/O scheduling priority of server processes. I/O niceness priority

[object-server]

Option	Default	Description
use		paste.deploy entry point for the object server. For most cases, this should be `egg:swift#object`.
set log_name	object-server	Label used when logging
set log_facility	LOG_LOCAL0	Syslog log facility
set log_level	INFO	Logging level
set log_requests	True	Whether or not to log each request
set log_address	/dev/log	Logging directory
user	swift	User to run as
max_upload_time	86400	Maximum time allowed to upload an object
slow	0	If > 0, Minimum time in seconds for a PUT or DELETE request to complete. This is only useful to simulate slow devices during testing and development.
mb_per_sync	512	On PUT requests, sync file every n MB
keep_cache_size	5242880	Largest object size to keep in buffer cache
keep_cache_private	false	Allow non-public objects to stay in kernels buffer cache
allowed_headers	Content-Disposition, Content-Encoding, X-Delete-At, X-Object-Manifest, X-Static-Large-Object Cache-Control, Content-Language, Expires, X-Robots-Tag	Comma separated list of headers that can be set in metadata on an object. This list is in addition to X-Object-Meta-* headers and cannot include Content-Type, etag, Content-Length, or deleted
replication_server		Configure parameter for creating specific server. To handle all verbs, including replication verbs, do not specify replication_server (this is the default). To only handle replication, set to a True value (e.g. True or 1). To handle only non-replication verbs, set to False. Unless you have a separate replication network, you should not specify any value for replication_server.
replication_concurrency	4	Set to restrict the number of concurrent incoming SSYNC requests; set to 0 for unlimited
replication_concurrency_per_device	1	Set to restrict the number of concurrent incoming SSYNC requests per device; set to 0 for unlimited requests per devices. This can help control I/O to each device. This does not override replication_concurrency described above, so you may need to adjust both parameters depending on your hardware or network capacity.
replication_lock_timeout	15	Number of seconds to wait for an existing replication device lock before giving up.
replication_failure_threshold	100	The number of subrequest failures before the replication_failure_ratio is checked
replication_failure_ratio	1.0	If the value of failures / successes of SSYNC subrequests exceeds this ratio, the overall SSYNC request will be aborted

[object-replicator]

Option	Default	Description
log_name	object-replicator	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
daemonize	yes	Whether or not to run replication as a daemon
interval	30	Time in seconds to wait between replication passes
concurrency	1	Number of replication jobs to run per worker process
replicator_workers	0	Number of worker processes to use. No matter how big this number is, at most one worker per disk will be used. The default value of 0 means no forking; all work is done in the main process.
sync_method	rsync	The sync method to use; default is rsync but you can use ssync to try the EXPERIMENTAL all-swift-code-no-rsync-callouts method. Once ssync is verified as or better than, rsync, we plan to deprecate rsync so we can move on with more features for replication.
rsync_timeout	900	Max duration of a partition rsync
rsync_bwlimit	0	Bandwidth limit for rsync in kB/s. 0 means unlimited.
rsync_io_timeout	30	Timeout value sent to rsync timeout and contimeout options
rsync_compress	no	Allow rsync to compress data which is transmitted to destination node during sync. However, this is applicable only when destination node is in a different region than the local one. NOTE: Objects that are already compressed (for example: .tar.gz, .mp3) might slow down the syncing process.
stats_interval	300	Interval in seconds between logging replication statistics
handoffs_first	false	If set to True, partitions that are not supposed to be on the node will be replicated first. The default setting should not be changed, except for extreme situations.
handoff_delete	auto	By default handoff partitions will be removed when it has successfully replicated to all the canonical nodes. If set to an integer n, it will remove the partition if it is successfully replicated to n nodes. The default setting should not be changed, except for extreme situations.
node_timeout	DEFAULT or 10	Request timeout to external services. This uses whats set here, or whats set in the DEFAULT section, or 10 (though other sections use 3 as the final default).
http_timeout	60	Max duration of an http request. This is for REPLICATE finalization calls and so should be longer than node_timeout.
lockup_timeout	1800	Attempts to kill all workers if nothing replicates for lockup_timeout seconds
rsync_module	{replication_ip}::object	Format of the rsync module where the replicator will send data. The configuration value can include some variables that will be extracted from the ring. Variables must follow the format {NAME} where NAME is one of: ip, port, replication_ip, replication_port, region, zone, device, meta. See etc/rsyncd.conf-sample for some examples.
rsync_error_log_line_length	0	Limit how long rsync error log lines are
ring_check_interval	15	Interval for checking new ring file
recon_cache_path	/var/cache/swift	Path to recon cache
nice_priority	None	Scheduling priority of server processes. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process). The default does not

[object-reconstructor]

Option	Default	Description
log_name	object-reconstructor	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
daemonize	yes	Whether or not to run reconstruction as a daemon
interval	30	Time in seconds to wait between reconstruction passes
reconstructor_workers	0	Maximum number of worker processes to spawn. Each worker will handle a subset of devices. Devices will be assigned evenly among the workers so that workers cycle at similar intervals (which can lead to fewer workers than requested). You can not have more workers than devices. If you have no devices only a single worker is spawned.
concurrency	1	Number of reconstruction threads to spawn per reconstructor process.
stats_interval	300	Interval in seconds between logging reconstruction statistics
handoffs_only	false	The handoffs_only mode option is for special case emergency situations during rebalance such as disk full in the cluster. This option SHOULD NOT BE CHANGED, except for extreme situations. When handoffs_only mode is enabled the reconstructor will *only* revert fragments from handoff nodes to primary nodes and will not sync primary nodes with neighboring primary nodes. This will force the reconstructor to sync and delete handoffs fragments more quickly and minimize the time of the rebalance by limiting the number of rebuilds. The handoffs_only option is only for temporary use and should be disabled as soon as the emergency situation has been resolved.
rebuild_handoff_node_count	2	The default strategy for unmounted drives will stage rebuilt data on a handoff node until updated rings are deployed. Because fragments are rebuilt on offset handoffs based on fragment index and the proxy limits how deep it will search for EC frags we restrict how many nodes well try. Setting to 0 will disable rebuilds to handoffs and only rebuild fragments for unmounted devices to mounted primaries after a ring change. Setting to -1 means no limit.
max_objects_per_revert	0	By default the reconstructor attempts to revert all objects from handoff partitions in a single batch using a single SSYNC request. In exceptional circumstances max_objects_per_revert can be used to temporarily limit the number of objects reverted by each reconstructor revert type job. If more than max_objects_per_revert are available in a senders handoff partition, the remaining objects will remain in the handoff partition and will not be reverted until the next time the reconstructor visits that handoff partition i.e. with this option set, a single cycle of the reconstructor may not completely revert all handoff partitions. The option has no effect on reconstructor sync type jobs between primary partitions. A value of 0 (the default) means there is no limit.
node_timeout	DEFAULT or 10	Request timeout to external services. The value used is the value set in this section, or the value set in the DEFAULT section, or 10.
http_timeout	60	Max duration of an http request. This is for REPLICATE finalization calls and so should be longer than node_timeout.
lockup_timeout	1800	Attempts to kill all threads if no fragment has been reconstructed for lockup_timeout seconds.
ring_check_interval	15	Interval for checking new ring file
recon_cache_path	/var/cache/swift	Path to recon cache

[object-updater]

Option	Default	Description
log_name	object-updater	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
interval	300	Minimum time for a pass to take
updater_workers	1	Number of worker processes
concurrency	8	Number of updates to run concurrently in each worker process
node_timeout	DEFAULT or 10	Request timeout to external services. This uses whats set here, or whats set in the DEFAULT section, or 10 (though other sections use 3 as the final default).
objects_per_second	50	Maximum objects updated per second. Should be tuned according to individual system specs. 0 is unlimited.
slowdown	0.01	Time in seconds to wait between objects. Deprecated in favor of objects_per_second.
report_interval	300	Interval in seconds between logging statistics about the current update pass.
recon_cache_path	/var/cache/swift	Path to recon cache
nice_priority	None	Scheduling priority of server processes. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process). The default does not modify priority.
ionice_class	None	I/O scheduling class of server processes. I/O niceness class values are IOPRIO_CLASS_RT (realtime), IOPRIO_CLASS_BE (best-effort), and IOPRIO_CLASS_IDLE (idle). The default does not modify class and priority. Linux supports io scheduling priorities and classes since 2.6.13 with the CFQ io scheduler. Work only with ionice_priority.
ionice_priority	None	I/O scheduling priority of server processes. I/O niceness priority is a number which goes from 0 to 7. The higher the value, the lower the I/O priority of the process. Work only with ionice_class. Ignored if IOPRIO_CLASS_IDLE is set.

[object-auditor]

Option	Default	Description
log_name	object-auditor	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
log_time	3600	Frequency of status logs in seconds.
interval	30	Time in seconds to wait between auditor passes
disk_chunk_size	65536	Size of chunks read during auditing
files_per_second	20	Maximum files audited per second per auditor process. Should be tuned according to individual system specs. 0 is unlimited.
bytes_per_second	10000000	Maximum bytes audited per second per auditor process. Should be tuned according to individual system specs. 0 is unlimited.
concurrency	1	The number of parallel processes to use for checksum auditing.
zero_byte_files_per_second	50	
object_size_stats		
recon_cache_path	/var/cache/swift	Path to recon cache
rsync_tempfile_timeout	auto	Time elapsed in seconds before rsync tempfiles will be unlinked. Config value of auto try to use object-replicators rsync_timeout + 900 or fallback to 86400 (1 day).
nice_priority	None	Scheduling priority of server processes. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process). The default does not modify priority.
ionice_class	None	I/O scheduling class of server processes. I/O niceness class values are IOPRIO_CLASS_RT (realtime), IOPRIO_CLASS_BE (best-effort), and IOPRIO_CLASS_IDLE (idle). The default does not modify class and priority. Linux supports io scheduling priorities and classes since 2.6.13 with the CFQ io scheduler. Work only with ionice_priority.
ionice_priority	None	I/O scheduling priority of server processes. I/O niceness priority is a number which goes from 0 to 7. The higher the value, the lower the I/O priority of the process. Work only with ionice_class. Ignored if IOPRIO_CLASS_IDLE is set.

[object-expirer]

Option	Default	Description
log_name	object-expirer	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_address	/dev/log	Logging directory
interval	300	Time in seconds to wait between expirer passes
report_interval	300	Frequency of status logs in seconds.
concurrency	1	Level of concurrency to use to do the work, this value must be set to at least 1
expiring_objects_account_name	expiring_objects	name for legacy expirer task queue
dequeue_from_legacy	False	This service will look for jobs on the legacy expirer task queue.
processes	0	How many parts to divide the legacy work into, one part per process that will be doing the work. When set 0 means that a single legacy process will be doing all the work. This can only be used in conjunction with `dequeue_from_legacy`.
process	0	Which of the parts a particular legacy process will work on. It is zero based, if you want to use 3 processes, you should run processes with process set to 0, 1, and 2. This can only be used in conjunction with `dequeue_from_legacy`.
reclaim_age	604800	How long an un-processable expired object marker will be retried before it is abandoned. It is not coupled with the tombstone reclaim age in the consistency engine.
request_tries	3	The number of times the expirers internal client will attempt any given request in the event of failure
recon_cache_path	/var/cache/swift	Path to recon cache
nice_priority	None	Scheduling priority of server processes. Niceness values range from -20 (most favorable to the process) to 19 (least favorable to the process). The default does not modify priority.
ionice_class	None	I/O scheduling class of server processes. I/O niceness class values are IOPRIO_CLASS_RT (realtime), IOPRIO_CLASS_BE (best-effort), and IOPRIO_CLASS_IDLE (idle). The default does not modify class and priority. Linux supports io scheduling priorities and classes since 2.6.13 with the CFQ io scheduler. Work only with ionice_priority.
ionice_priority	None	I/O scheduling priority of server processes. I/O niceness priority is a number which goes from 0 to 7. The higher the value, the lower the I/O priority of the process. Work only with ionice_class. Ignored if IOPRIO_CLASS_IDLE is set.

Configuration options for middleware can be found at:

- *Middleware*
- *The Auth System*

CHAPTER

SIX

OBJECT STORAGE V1 REST API DOCUMENTATION

See Complete Reference for the Object Storage REST API

The following provides supporting information for the REST API:

6.1 Discoverability

Your Object Storage system might not enable all features that you read about because your service provider chooses which features to enable.

To discover which features are enabled in your Object Storage system, use the `/info` request. However, your service provider might have disabled the `/info` request, or you might be using an older version that does not support the `/info` request.

To use the `/info` request, send a **GET** request using the `/info` path to the Object Store endpoint as shown in this example:

```
# curl https://storage.clouddrive.com/info
```

This example shows a truncated response body:

```
{
   "swift":{
      "version":"1.11.0"
   },
   "staticweb":{

   },
   "tempurl":{

   }
}
```

This output shows that the Object Storage system has enabled the static website and temporary URL features.

6.2 Authentication

The owner of an Object Storage account controls access to that account and its containers and objects. An owner is the user who has the admin role for that tenant. The tenant is also known as the project or account. As the account owner, you can modify account metadata and create, modify, and delete containers and objects.

To identify yourself as the account owner, include an authentication token in the X-Auth-Token header in the API request.

Depending on the token value in the X-Auth-Token header, one of the following actions occur:

- X-Auth-Token contains the token for the account owner.

  The request is permitted and has full access to make changes to the account.

- The X-Auth-Token header is omitted or it contains a token for a non-owner or a token that is not valid.

  The request fails with a 401 Unauthorized or 403 Forbidden response.

  You have no access to accounts or containers, unless an access control list (ACL) explicitly grants access.

  The account owner can grant account and container access to users through access control lists (ACLs).

In addition, it is possible to provide an additional token in the X-Service-Token header. More information about how this is used is in *Using Swift as Backing Store for Service Data*.

The following list describes the authentication services that you can use with Object Storage:

- OpenStack Identity (keystone): For Object Storage, account is synonymous with project or tenant ID.
- Tempauth middleware: Object Storage includes this middleware. User and account management is performed in Object Storage itself.
- Swauth middleware: Stored in github, this custom middleware is modeled on Tempauth. Usage is similar to Tempauth.
- Other custom middleware: Write it yourself to fit your environment.

Specifically, you use the X-Auth-Token header to pass an authentication token to an API request.

Authentication tokens expire after a time period that the authentication service defines. When a token expires, use of the token causes requests to fail with a 401 Unauthorized response. To continue, you must obtain a new token.

6.3 Container quotas

You can set quotas on the size and number of objects stored in a container by setting the following metadata:

- `X-Container-Meta-Quota-Bytes`. The size, in bytes, of objects that can be stored in a container.
- `X-Container-Meta-Quota-Count`. The number of objects that can be stored in a container.

When you exceed a container quota, subsequent requests to create objects fail with a 413 Request Entity Too Large error.

The Object Storage system uses an eventual consistency model. When you create a new object, the container size and object count might not be immediately updated. Consequently, you might be allowed to create objects even though you have actually exceeded the quota.

At some later time, the system updates the container size and object count to the actual values. At this time, subsequent requests fails. In addition, if you are currently under the `X-Container-Meta-Quota-Bytes` limit and a request uses chunked transfer encoding, the system cannot know if the request will exceed the quota so the system allows the request. However, once the quota is exceeded, any subsequent uploads that use chunked transfer encoding fail.

6.4 Object versioning

You can store multiple versions of your content so that you can recover from unintended overwrites. Object versioning is an easy way to implement version control, which you can use with any type of content.

Note: You cannot version a large-object manifest file, but the large-object manifest file can point to versioned segments.

Note: It is strongly recommended that you put non-current objects in a different container than the container where current object versions reside.

To allow object versioning within a cluster, the cloud provider should add the `versioned_writes` filter to the pipeline and set the `allow_versioned_writes` option to `true` in the `[filter:versioned_writes]` section of the proxy-server configuration file.

To enable object versioning for a container, you must specify an archive container that will retain non-current versions via either the `X-Versions-Location` or `X-History-Location` header. These two headers enable two distinct modes of operation. Either mode may be used within a cluster, but only one mode may be active for any given container. You must UTF-8-encode and then URL-encode the container name before you include it in the header.

For both modes, **PUT** requests will archive any pre-existing objects before writing new data, and **GET** requests will serve the current version. **COPY** requests behave like a **GET** followed by a **PUT**; that is, if the copy *source* is in a versioned container then the current version will be copied, and if the copy *destination* is in a versioned container then any pre-existing object will be archived before writing new data.

If object versioning was enabled using `X-History-Location`, then object **DELETE** requests will copy the current version to the archive container then remove it from the versioned container.

If object versioning was enabled using `X-Versions-Location`, then object **DELETE** requests will restore the most-recent version from the archive container, overwriting the current version.

6.4.1 Example Using `X-Versions-Location`

1. Create the `current` container:

```
# curl -i $publicURL/current -X PUT -H "Content-Length: 0" -H "X-Auth-
↪Token: $token" -H "X-Versions-Location: archive"
```

```
HTTP/1.1 201 Created
Content-Length: 0
Content-Type: text/html; charset=UTF-8
X-Trans-Id: txb91810fb717347d09eec8-0052e18997
X-Openstack-Request-Id: txb91810fb717347d09eec8-0052e18997
Date: Thu, 23 Jan 2014 21:28:55 GMT
```

2. Create the first version of an object in the `current` container:

```
# curl -i $publicURL/current/my_object --data-binary 1 -X PUT -H "Content-
↪Length: 0" -H "X-Auth-Token: $token"
```

```
HTTP/1.1 201 Created
Last-Modified: Thu, 23 Jan 2014 21:31:22 GMT
Content-Length: 0
Etag: d41d8cd98f00b204e9800998ecf8427e
Content-Type: text/html; charset=UTF-8
X-Trans-Id: tx5992d536a4bd4fec973aa-0052e18a2a
X-Openstack-Request-Id: tx5992d536a4bd4fec973aa-0052e18a2a
Date: Thu, 23 Jan 2014 21:31:22 GMT
```

   Nothing is written to the non-current version container when you initially **PUT** an object in the `current` container. However, subsequent **PUT** requests that edit an object trigger the creation of a version of that object in the `archive` container.

   These non-current versions are named as follows:

```
<length><object_name>/<timestamp>
```

   Where `length` is the 3-character, zero-padded hexadecimal character length of the object, `<object_name>` is the object name, and `<timestamp>` is the time when the object was initially created as a current version.

3. Create a second version of the object in the `current` container:

```
# curl -i $publicURL/current/my_object --data-binary 2 -X PUT -H "Content-
↪Length: 0" -H "X-Auth-Token: $token"
```

```
HTTP/1.1 201 Created
Last-Modified: Thu, 23 Jan 2014 21:41:32 GMT
Content-Length: 0
Etag: d41d8cd98f00b204e9800998ecf8427e
Content-Type: text/html; charset=UTF-8
X-Trans-Id: tx468287ce4fc94eada96ec-0052e18c8c
X-Openstack-Request-Id: tx468287ce4fc94eada96ec-0052e18c8c
Date: Thu, 23 Jan 2014 21:41:32 GMT
```

4. Issue a **GET** request to a versioned object to get the current version of the object. You do not have to do any request redirects or metadata lookups.

   List older versions of the object in the `archive` container:

```
# curl -i $publicURL/archive?prefix=009my_object -X GET -H "X-Auth-Token:
↪$token"
```

```
HTTP/1.1 200 OK
Content-Length: 30
X-Container-Object-Count: 1
Accept-Ranges: bytes
X-Timestamp: 1390513280.79684
X-Container-Bytes-Used: 0
Content-Type: text/plain; charset=utf-8
X-Trans-Id: tx9a441884997542d3a5868-0052e18d8e
X-Openstack-Request-Id: tx9a441884997542d3a5868-0052e18d8e
Date: Thu, 23 Jan 2014 21:45:50 GMT

009my_object/1390512682.92052
```

Note: A **POST** request to a versioned object updates only the metadata for the object and does not create a new version of the object. New versions are created only when the content of the object changes.

---

5. Issue a **DELETE** request to a versioned object to remove the current version of the object and replace it with the next-most current version in the non-current container.

```
# curl -i $publicURL/current/my_object -X DELETE -H "X-Auth-Token: $token"
```

```
HTTP/1.1 204 No Content
Content-Length: 0
Content-Type: text/html; charset=UTF-8
X-Trans-Id: tx006d944e02494e229b8ee-0052e18edd
X-Openstack-Request-Id: tx006d944e02494e229b8ee-0052e18edd
Date: Thu, 23 Jan 2014 21:51:25 GMT
```

   List objects in the `archive` container to show that the archived object was moved back to the `current` container:

```
# curl -i $publicURL/archive?prefix=009my_object -X GET -H "X-Auth-Token:
↪$token"
```

```
HTTP/1.1 204 No Content
Content-Length: 0
X-Container-Object-Count: 0
Accept-Ranges: bytes
X-Timestamp: 1390513280.79684
X-Container-Bytes-Used: 0
Content-Type: text/html; charset=UTF-8
X-Trans-Id: tx044f2a05f56f4997af737-0052e18eed
X-Openstack-Request-Id: tx044f2a05f56f4997af737-0052e18eed
Date: Thu, 23 Jan 2014 21:51:41 GMT
```

This next-most current version carries with it any metadata last set on it. If want to completely remove an object and you have five versions of it, you must **DELETE** it five times.

6.4.2 Example Using `X-History-Location`

1. Create the `current` container:

```
# curl -i $publicURL/current -X PUT -H "Content-Length: 0" -H "X-Auth-
↪Token: $token" -H "X-History-Location: archive"
```

```
HTTP/1.1 201 Created
Content-Length: 0
Content-Type: text/html; charset=UTF-8
X-Trans-Id: txb91810fb717347d09eec8-0052e18997
X-Openstack-Request-Id: txb91810fb717347d09eec8-0052e18997
Date: Thu, 23 Jan 2014 21:28:55 GMT
```

2. Create the first version of an object in the `current` container:

```
# curl -i $publicURL/current/my_object --data-binary 1 -X PUT -H "Content-
↪Length: 0" -H "X-Auth-Token: $token"
```

```
HTTP/1.1 201 Created
Last-Modified: Thu, 23 Jan 2014 21:31:22 GMT
Content-Length: 0
Etag: d41d8cd98f00b204e9800998ecf8427e
Content-Type: text/html; charset=UTF-8
X-Trans-Id: tx5992d536a4bd4fec973aa-0052e18a2a
X-Openstack-Request-Id: tx5992d536a4bd4fec973aa-0052e18a2a
Date: Thu, 23 Jan 2014 21:31:22 GMT
```

Nothing is written to the non-current version container when you initially **PUT** an object in the `current` container. However, subsequent **PUT** requests that edit an object trigger the creation of a version of that object in the `archive` container.

These non-current versions are named as follows:

```
<length><object_name>/<timestamp>
```

Where `length` is the 3-character, zero-padded hexadecimal character length of the object, `<object_name>` is the object name, and `<timestamp>` is the time when the object was initially created as a current version.

3. Create a second version of the object in the `current` container:

```
# curl -i $publicURL/current/my_object --data-binary 2 -X PUT -H "Content-
↪Length: 0" -H "X-Auth-Token: $token"
```

```
HTTP/1.1 201 Created
Last-Modified: Thu, 23 Jan 2014 21:41:32 GMT
Content-Length: 0
Etag: d41d8cd98f00b204e9800998ecf8427e
Content-Type: text/html; charset=UTF-8
X-Trans-Id: tx468287ce4fc94eada96ec-0052e18c8c
X-Openstack-Request-Id: tx468287ce4fc94eada96ec-0052e18c8c
Date: Thu, 23 Jan 2014 21:41:32 GMT
```

4. Issue a **GET** request to a versioned object to get the current version of the object. You do not have to do any request redirects or metadata lookups.

   List older versions of the object in the `archive` container:

```
# curl -i $publicURL/archive?prefix=009my_object -X GET -H "X-Auth-Token:
↪$token"
```

```
HTTP/1.1 200 OK
Content-Length: 30
X-Container-Object-Count: 1
Accept-Ranges: bytes
X-Timestamp: 1390513280.79684
X-Container-Bytes-Used: 0
Content-Type: text/plain; charset=utf-8
X-Trans-Id: tx9a441884997542d3a5868-0052e18d8e
X-Openstack-Request-Id: tx9a441884997542d3a5868-0052e18d8e
Date: Thu, 23 Jan 2014 21:45:50 GMT

009my_object/1390512682.92052
```

Note: A **POST** request to a versioned object updates only the metadata for the object and does not create a new version of the object. New versions are created only when the content of the object changes.

---

5. Issue a **DELETE** request to a versioned object to copy the current version of the object to the archive container then delete it from the current container. Subsequent **GET** requests to the object in the current container will return `404 Not Found`.

```
# curl -i $publicURL/current/my_object -X DELETE -H "X-Auth-Token: $token"
```

```
HTTP/1.1 204 No Content
Content-Length: 0
Content-Type: text/html; charset=UTF-8
X-Trans-Id: tx006d944e02494e229b8ee-0052e18edd
X-Openstack-Request-Id: tx006d944e02494e229b8ee-0052e18edd
Date: Thu, 23 Jan 2014 21:51:25 GMT
```

List older versions of the object in the `archive` container:

```
# curl -i $publicURL/archive?prefix=009my_object -X GET -H "X-Auth-Token:
↪$token"
```

```
HTTP/1.1 200 OK
Content-Length: 90
X-Container-Object-Count: 3
Accept-Ranges: bytes
X-Timestamp: 1390513280.79684
X-Container-Bytes-Used: 0
Content-Type: text/html; charset=UTF-8
X-Trans-Id: tx044f2a05f56f4997af737-0052e18eed
X-Openstack-Request-Id: tx044f2a05f56f4997af737-0052e18eed
Date: Thu, 23 Jan 2014 21:51:41 GMT

009my_object/1390512682.92052
009my_object/1390512692.23062
009my_object/1390513885.67732
```

In addition to the two previous versions of the object, the archive container has a delete marker to record when the object was deleted.

To permanently delete a previous version, issue a **DELETE** to the version in the archive container.

6.4.3 Disabling Object Versioning

To disable object versioning for the `current` container, remove its `X-Versions-Location` metadata header by sending an empty key value.

```
# curl -i $publicURL/current -X PUT -H "Content-Length: 0" -H "X-Auth-Token:
↪$token" -H "X-Versions-Location: "
```

```
HTTP/1.1 202 Accepted
Content-Length: 76
Content-Type: text/html; charset=UTF-8
X-Trans-Id: txe2476de217134549996d0-0052e19038
X-Openstack-Request-Id: txe2476de217134549996d0-0052e19038
Date: Thu, 23 Jan 2014 21:57:12 GMT

<html><h1>Accepted</h1><p>The request is accepted for processing.</p></html>
```

6.5 Large objects

By default, the content of an object cannot be greater than 5 GB. However, you can use a number of smaller objects to construct a large object. The large object is comprised of two types of objects:

- **Segment objects** store the object content. You can divide your content into segments, and upload each segment into its own segment object. Segment objects do not have any special features. You create, update, download, and delete segment objects just as you would normal objects.
- A **manifest object** links the segment objects into one logical large object. When you download a manifest object, Object Storage concatenates and returns the contents of the segment objects in the response body of the request. This behavior extends to the response headers returned by **GET** and **HEAD** requests. The `Content-Length` response header value is the total size of all segment objects. Object Storage calculates the `ETag` response header value by taking the `ETag` value of each segment, concatenating them together, and returning the MD5 checksum of the result. The manifest object types are:

  Static large objects The manifest object content is an ordered list of the names of the segment objects in JSON format.

  Dynamic large objects The manifest object has a `X-Object-Manifest` metadata header. The value of this header is `{container}/{prefix}`, where `{container}` is the name of the container where the segment objects are stored, and `{prefix}` is a string that all segment objects have in common. The manifest object should have no content. However, this is not enforced.

6.5.1 Note

If you make a **COPY** request by using a manifest object as the source, the new object is a normal, and not a segment, object. If the total size of the source segment objects exceeds 5 GB, the **COPY** request fails. However, you can make a duplicate of the manifest object and this new object can be larger than 5 GB.

6.5.2 Static large objects

To create a static large object, divide your content into pieces and create (upload) a segment object to contain each piece.

Create a manifest object. Include the `multipart-manifest=put` query parameter at the end of the manifest object name to indicate that this is a manifest object.

The body of the **PUT** request on the manifest object comprises a json list, where each element is an object representing a segment. These objects may contain the following attributes:

- `path` (required). The container and object name in the format: `{container-name}/{object-name}`
- `etag` (optional). If provided, this value must match the `ETag` of the segment object. This was included in the response headers when the segment was created. Generally, this will be the MD5 sum of the segment.
- `size_bytes` (optional). The size of the segment object. If provided, this value must match the `Content-Length` of that object.

- `range` (optional). The subset of the referenced object that should be used for segment data. This behaves similar to the `Range` header. If omitted, the entire object will be used.

Providing the optional `etag` and `size_bytes` attributes for each segment ensures that the upload cannot corrupt your data.

Example Static large object manifest list

This example shows three segment objects. You can use several containers and the object names do not have to conform to a specific pattern, in contrast to dynamic large objects.

```
[
    {
        "path": "mycontainer/objseg1",
        "etag": "0228c7926b8b642dfb29554cd1f00963",
        "size_bytes": 1468006
    },
    {
        "path": "mycontainer/pseudodir/seg-obj2",
        "etag": "5bfc9ea51a00b790717eeb934fb77b9b",
        "size_bytes": 1572864
    },
    {
        "path": "other-container/seg-final",
        "etag": "b9c3da507d2557c1ddc51f27c54bae51",
        "size_bytes": 256
    }
]
```

The `Content-Length` request header must contain the length of the json contentnot the length of the segment objects. However, after the **PUT** operation completes, the `Content-Length` metadata is set to the total length of all the object segments. When using the `ETag` request header in a **PUT** operation, it must contain the MD5 checksum of the concatenated `ETag` values of the object segments. You can also set the `Content-Type` request header and custom object metadata.

When the **PUT** operation sees the `multipart-manifest=put` query parameter, it reads the request body and verifies that each segment object exists and that the sizes and ETags match. If there is a mismatch, the **PUT** operation fails.

This verification process can take a long time to complete, particularly as the number of segments increases. You may include a `heartbeat=on` query parameter to have the server:

1. send a `202 Accepted` response before it begins validating segments,
2. periodically send whitespace characters to keep the connection alive, and
3. send a final response code in the body.

Note: The server may still immediately respond with `400 Bad Request` if it can determine that the

request is invalid before making backend requests.

If everything matches, the manifest object is created. The `X-Static-Large-Object` metadata is set to `true` indicating that this is a static object manifest.

Normally when you perform a **GET** operation on the manifest object, the response body contains the concatenated content of the segment objects. To download the manifest list, use the `multipart-manifest=get` query parameter. The resulting list is not formatted the same as the manifest you originally used in the **PUT** operation.

If you use the **DELETE** operation on a manifest object, the manifest object is deleted. The segment objects are not affected. However, if you add the `multipart-manifest=delete` query parameter, the segment objects are deleted and if all are successfully deleted, the manifest object is also deleted.

To change the manifest, use a **PUT** operation with the `multipart-manifest=put` query parameter. This request creates a manifest object. You can also update the object metadata in the usual way.

6.5.3 Dynamic large objects

You must segment objects that are larger than 5 GB before you can upload them. You then upload the segment objects like you would any other object and create a dynamic large manifest object. The manifest object tells Object Storage how to find the segment objects that comprise the large object. The segments remain individually addressable, but retrieving the manifest object streams all the segments concatenated. There is no limit to the number of segments that can be a part of a single large object, but `Content-Length` is included in **GET** or **HEAD** response only if the number of segments is smaller than container listing limit. In other words, the number of segments that fit within a single container listing page.

To ensure the download works correctly, you must upload all the object segments to the same container and ensure that each object name is prefixed in such a way that it sorts in the order in which it should be concatenated. You also create and upload a manifest file. The manifest file is a zero-byte file with the extra `X-Object-Manifest {container}/{prefix}` header, where `{container}` is the container the object segments are in and `{prefix}` is the common prefix for all the segments. You must UTF-8-encode and then URL-encode the container and common prefix in the `X-Object-Manifest` header.

It is best to upload all the segments first and then create or update the manifest. With this method, the full object is not available for downloading until the upload is complete. Also, you can upload a new set of segments to a second location and update the manifest to point to this new location. During the upload of the new segments, the original manifest is still available to download the first set of segments.

Note: When updating a manifest object using a POST request, a `X-Object-Manifest` header must be included for the object to continue to behave as a manifest object.

Example Upload segment of large object request: HTTP

```
PUT /{api_version}/{account}/{container}/{object} HTTP/1.1
Host: storage.clouddrive.com
X-Auth-Token: eaaafd18-0fed-4b3a-81b4-663c99ec1cbb
ETag: 8a964ee2a5e88be344f36c22562a6486
Content-Length: 1
X-Object-Meta-PIN: 1234
```

No response body is returned. A status code of 2*"nn"* (between 200 and 299, inclusive) indicates a successful write; status 411 Length Required denotes a missing `Content-Length` or `Content-Type` header in the request. If the MD5 checksum of the data written to the storage system does NOT match the (optionally) supplied ETag value, a 422 Unprocessable Entity response is returned.

You can continue uploading segments like this example shows, prior to uploading the manifest.

Example Upload next segment of large object request: HTTP

```
PUT /{api_version}/{account}/{container}/{object} HTTP/1.1
Host: storage.clouddrive.com
X-Auth-Token: eaaafd18-0fed-4b3a-81b4-663c99ec1cbb
ETag: 8a964ee2a5e88be344f36c22562a6486
Content-Length: 1
X-Object-Meta-PIN: 1234
```

Next, upload the manifest you created that indicates the container the object segments reside within. Note that uploading additional segments after the manifest is created causes the concatenated object to be that much larger but you do not need to recreate the manifest file for subsequent additional segments.

Example Upload manifest request: HTTP

```
PUT /{api_version}/{account}/{container}/{object} HTTP/1.1
Host: storage.clouddrive.com
X-Auth-Token: eaaafd18-0fed-4b3a-81b4-663c99ec1cbb
Content-Length: 0
X-Object-Meta-PIN: 1234
X-Object-Manifest: {container}/{prefix}
```

Example Upload manifest response: HTTP

```
[...]
```

The `Content-Type` in the response for a **GET** or **HEAD** on the manifest is the same as the `Content-Type` set during the **PUT** request that created the manifest. You can easily change the `Content-Type` by reissuing the **PUT** request.

6.5.4 Comparison of static and dynamic large objects

While static and dynamic objects have similar behavior, here are their differences:

End-to-end integrity

With static large objects, integrity can be assured. The list of segments may include the MD5 checksum (`ETag`) of each segment. You cannot upload the manifest object if the `ETag` in the list differs from the uploaded segment object. If a segment is somehow lost, an attempt to download the manifest object results in an error.

With dynamic large objects, integrity is not guaranteed. The eventual consistency model means that although you have uploaded a segment object, it might not appear in the container listing until later. If you download the manifest before it appears in the container, it does not form part of the content returned in response to a **GET** request.

Upload Order

With static large objects, you must upload the segment objects before you upload the manifest object.

With dynamic large objects, you can upload manifest and segment objects in any order. In case a premature download of the manifest occurs, we recommend users upload the manifest object after the segments. However, the system does not enforce the order.

Removal or addition of segment objects

With static large objects, you cannot add or remove segment objects from the manifest. However, you can create a completely new manifest object of the same name with a different manifest list.

With dynamic large objects, you can upload new segment objects or remove existing segments. The names must simply match the `{prefix}` supplied in `X-Object-Manifest`.

Segment object size and number

With static large objects, the segment objects must be at least 1 byte in size. However, if the segment objects are less than 1MB (by default), the SLO download is (by default) rate limited. At most, 1000 segments are supported (by default) and the manifest has a limit (by default) of 2MB in size.

With dynamic large objects, segment objects can be any size.

Segment object container name

With static large objects, the manifest list includes the container name of each object. Segment objects can be in different containers.

With dynamic large objects, all segment objects must be in the same container.

Manifest object metadata

With static large objects, the manifest object has `X-Static-Large-Object` set to `true`. You do not set this metadata directly. Instead the system sets it when you **PUT** a static manifest object.

With dynamic large objects, the `X-Object-Manifest` value is the `{container}/{prefix}`, which indicates where the segment objects are located. You supply this request header in the **PUT** operation.

Copying the manifest object

The semantics are the same for both static and dynamic large objects. When copying large objects, the **COPY** operation does not create a manifest object but a normal object with content same as what you would get on a **GET** request to the original manifest object.

To copy the manifest object, you include the `multipart-manifest=get` query parameter in the **COPY** request. The new object contains the same manifest as the original. The segment objects are not copied. Instead, both the original and new manifest objects share the same set of segment objects.

6.6 Temporary URL middleware

To discover whether your Object Storage system supports this feature, check with your service provider or send a **GET** request using the `/info` path.

A temporary URL gives users temporary access to objects. For example, a website might want to provide a link to download a large object in Object Storage, but the Object Storage account has no public access. The website can generate a URL that provides time-limited **GET** access to the object. When the web browser user clicks on the link, the browser downloads the object directly from Object Storage, eliminating the need for the website to act as a proxy for the request.

Furthermore, a temporary URL can be prefix-based. These URLs contain a signature which is valid for all objects which share a common prefix. They are useful for sharing a set of objects.

Ask your cloud administrator to enable the temporary URL feature. For information, see *TempURL* in the *Source Documentation*.

Note: To use **POST** requests to upload objects to specific Object Storage locations, use *Form POST middleware* instead of temporary URL middleware.

6.6.1 Temporary URL format

A temporary URL is comprised of the URL for an object with added query parameters:

Example Temporary URL format

```
https://swift-cluster.example.com/v1/my_account/container/object
?temp_url_sig=732fcac368abb10c78a4cbe95c3fab7f311584532bf779abd5074e13cbe8b88b
&temp_url_expires=1323479485
&filename=My+Test+File.pdf
```

The example shows these elements:

Object URL: Required. The full path URL to the object.

temp_url_sig: Required. An HMAC cryptographic signature that defines the allowed HTTP method, expiration date, full path to the object, and the secret key for the temporary URL. The digest used (for example, SHA-256 or SHA-512) must be supported by the cluster; supported digests will be listed in the `tempurl.allowed_digests` key in the clusters capabilities.

temp_url_expires: Required. An expiration date as a UNIX Epoch timestamp or ISO 8601 UTC timestamp. For example, `1390852007` or `2014-01-27T19:46:47Z` can be used to represent `Mon, 27 Jan 2014 19:46:47 GMT`.

For more information, see Epoch & Unix Timestamp Conversion Tools.

filename: Optional. Overrides the default file name. Object Storage generates a default file name for **GET** temporary URLs that is based on the object name. Object Storage returns this value in the `Content-Disposition` response header. Browsers can interpret this file name value as a file attachment to be saved.

A prefix-based temporary URL is similar but requires the parameter `temp_url_prefix`, which must be equal to the common prefix shared by all object names for which the URL is valid.

```
https://swift-cluster.example.com/v1/my_account/container/my_prefix/object
?temp_url_sig=732fcac368abb10c78a4cbe95c3fab7f311584532bf779abd5074e13cbe8b88b
&temp_url_expires=2011-12-10T01:11:25Z
&temp_url_prefix=my_prefix
```

6.6.2 Secret Keys

The cryptographic signature used in Temporary URLs and also in *Form POST middleware* uses a secret key. Object Storage allows you to store two secret key values per account, and two per container. When validating a request, Object Storage checks signatures against all keys. Using two keys at each level enables key rotation without invalidating existing temporary URLs.

To set the keys at the account level, set one or both of the following request headers to arbitrary values on a **POST** request to the account:

- `X-Account-Meta-Temp-URL-Key`
- `X-Account-Meta-Temp-URL-Key-2`

To set the keys at the container level, set one or both of the following request headers to arbitrary values on a **POST** or **PUT** request to the container:

- `X-Container-Meta-Temp-URL-Key`
- `X-Container-Meta-Temp-URL-Key-2`

The arbitrary values serve as the secret keys.

For example, use the **swift post** command to set the secret key to *"MYKEY"*:

```
$ swift post -m "Temp-URL-Key:MYKEY"
```

Note: Changing these headers invalidates any previously generated temporary URLs within 60 seconds, which is the memcache time for the key.

6.6.3 HMAC signature for temporary URLs

Temporary URL middleware uses an HMAC cryptographic signature. This signature includes these elements:

- The allowed method. Typically, **GET** or **PUT**.
- Expiry time. In the example for the HMAC-SHA256 signature for temporary URLs below, the expiry time is set to `86400` seconds (or 1 day) into the future. Please be aware that you have to use a UNIX timestamp for generating the signature (in the API request it is also allowed to use an ISO 8601 UTC timestamp).
- The path. Starting with `/v1/` onwards and including a container name and object. The path for prefix-based signatures must start with `prefix:/v1/`. Do not URL-encode the path at this stage.
- The secret key. Use one of the key values as described in *Secret Keys*.

These sample Python codes show how to compute a signature for use with temporary URLs:

Example HMAC-SHA256 signature for object-based temporary URLs

```
import hmac
from hashlib import sha256
from time import time
method = 'GET'
duration_in_seconds = 60*60*24
expires = int(time() + duration_in_seconds)
path = '/v1/my_account/container/object'
key = 'MYKEY'
hmac_body = '%s\n%s\n%s' % (method, expires, path)
signature = hmac.new(key, hmac_body, sha256).hexdigest()
```

Example HMAC-SHA512 signature for prefix-based temporary URLs

```
import hmac
from hashlib import sha512
from time import time
method = 'GET'
duration_in_seconds = 60*60*24
expires = int(time() + duration_in_seconds)
path = 'prefix:/v1/my_account/container/my_prefix'
key = 'MYKEY'
hmac_body = '%s\n%s\n%s' % (method, expires, path)
signature = hmac.new(key, hmac_body, sha512).hexdigest()
```

Do not URL-encode the path when you generate the HMAC signature. However, when you make the actual HTTP request, you should properly URL-encode the URL.

The *"MYKEY"* value is one of the key values as described in *Secret Keys*.

For more information, see RFC 2104: HMAC: Keyed-Hashing for Message Authentication.

If you want to transform a UNIX timestamp into an ISO 8601 UTC timestamp, you can use following code snippet:

```
import time
time.strftime('%Y-%m-%dT%H:%M:%SZ', time.gmtime(timestamp))
```

6.6.4 Using the `swift` tool to generate a Temporary URL

The `swift` tool provides the tempurl option that auto-generates the *"temp_url_sig"* and *"temp_url_expires"* query parameters. For example, you might run this command:

```
$ swift tempurl GET 3600 /v1/my_account/container/object MYKEY
```

Note: The `swift` tool is not yet updated and continues to use the deprecated cipher SHA1.

This command returns the path:

```
/v1/my_account/container/object
?temp_url_sig=5c4cc8886f36a9d0919d708ade98bf0cc71c9e91
&temp_url_expires=1374497657
```

To create the temporary URL, prefix this path with the Object Storage storage host name. For example, prefix the path with `https://swift-cluster.example.com`, as follows:

```
https://swift-cluster.example.com/v1/my_account/container/object
?temp_url_sig=5c4cc8886f36a9d0919d708ade98bf0cc71c9e91
&temp_url_expires=1374497657
```

Note that if the above example is copied exactly, and used in a command shell, then the ampersand is interpreted as an operator and the URL will be truncated. Enclose the URL in quotation marks to avoid this.

6.7 Form POST middleware

To discover whether your Object Storage system supports this feature, check with your service provider or send a **GET** request using the `/info` path.

You can upload objects directly to the Object Storage system from a browser by using the form **POST** middleware. This middleware uses account or container secret keys to generate a cryptographic signature for the request. This means that you do not need to send an authentication token in the `X-Auth-Token` header to perform the request.

The form **POST** middleware uses the same secret keys as the temporary URL middleware uses. For information about how to set these keys, see *Secret Keys*.

For information about the form **POST** middleware configuration options, see *FormPost* in the *Source Documentation*.

6.7.1 Form POST format

To upload objects to a cluster, you can use an HTML form **POST** request.

The format of the form **POST** request is:

Example 1.14. Form POST format

```
<form action="SWIFT_URL"
    method="POST"
    enctype="multipart/form-data">
    <input type="hidden" name="redirect" value="REDIRECT_URL"/>
    <input type="hidden" name="max_file_size" value="BYTES"/>
    <input type="hidden" name="max_file_count" value="COUNT"/>
    <input type="hidden" name="expires" value="UNIX_TIMESTAMP"/>
    <input type="hidden" name="signature" value="HMAC"/>
    <input type="file" name="FILE_NAME"/>
    <br/>
    <input type="submit"/>
</form>
```

action=SWIFT_URL

Set to full URL where the objects are to be uploaded. The names of uploaded files are appended to the specified *SWIFT_URL*. So, you can upload directly to the root of a container with a URL like:

```
https://swift-cluster.example.com/v1/my_account/container/
```

Optionally, you can include an object prefix to separate uploads, such as:

```
https://swift-cluster.example.com/v1/my_account/container/OBJECT_PREFIX
```

method=POST

Must be `POST`.

enctype=multipart/form-data

Must be `multipart/form-data`.

name=redirect value=REDIRECT_URL

Redirects the browser to the *REDIRECT_URL* after the upload completes. The URL has status and message query parameters added to it, which specify the HTTP status code for the upload and an optional error message. The 2*nn* status code indicates success.

The *REDIRECT_URL* can be an empty string. If so, the `Location` response header is not set.

name=max_file_size value=BYTES

Required. Indicates the size, in bytes, of the maximum single file upload.

name=max_file_count value= COUNT

Required. Indicates the maximum number of files that can be uploaded with the form.

name=expires value=UNIX_TIMESTAMP

The UNIX timestamp that specifies the time before which the form must be submitted before it becomes no longer valid.

name=signature value=HMAC

The HMAC-SHA1 signature of the form.

type=file name=FILE_NAME

File name of the file to be uploaded. You can include from one to the `max_file_count` value of files.

The file attributes must appear after the other attributes to be processed correctly.

If attributes appear after the file attributes, they are not sent with the sub-request because all attributes in the file cannot be parsed on the server side unless the whole file is read into memory; the server does not have enough memory to service these requests. Attributes that follow the file attributes are ignored.

Optionally, if you want the uploaded files to be temporary you can set x-delete-at or x-delete-after attributes by adding one of these as a form input:

```
<input type="hidden" name="x_delete_at" value="<unix-timestamp>" />
<input type="hidden" name="x_delete_after" value="<seconds>" />
```

type= submit

Must be `submit`.

6.7.2 HMAC-SHA1 signature for form POST

Form **POST** middleware uses an HMAC-SHA1 cryptographic signature. This signature includes these elements from the form:

- The path. Starting with `/v1/` onwards and including a container name and, optionally, an object prefix. In *Example 1.15*, HMAC-SHA1 signature for form POST the path is `/v1/my_account/container/object_prefix`. Do not URL-encode the path at this stage.
- A redirect URL. If there is no redirect URL, use the empty string.
- Maximum file size. In *Example 1.15*, HMAC-SHA1 signature for form POST the `max_file_size` is `104857600` bytes.
- The maximum number of objects to upload. In *Example 1.15*, HMAC-SHA1 signature for form POST `max_file_count` is `10`.
- Expiry time. In *Example 1.15, HMAC-SHA1 signature for form POST the expiry time is set to "600"* seconds into the future.
- The secret key. Set as the `X-Account-Meta-Temp-URL-Key` header value for accounts or `X-Container-Meta-Temp-URL-Key` header value for containers. See *Secret Keys* for more information.

The following example code generates a signature for use with form **POST**:

Example 1.15. HMAC-SHA1 signature for form POST

```
import hmac
from hashlib import sha1
from time import time
path = '/v1/my_account/container/object_prefix'
redirect = 'https://myserver.com/some-page'
max_file_size = 104857600
max_file_count = 10
expires = int(time() + 600)
key = 'MYKEY'
hmac_body = '%s\n%s\n%s\n%s\n%s' % (path, redirect,
max_file_size, max_file_count, expires)
signature = hmac.new(key, hmac_body, sha1).hexdigest()
```

For more information, see RFC 2104: HMAC: Keyed-Hashing for Message Authentication.

6.7.3 Form POST example

The following example shows how to submit a form by using a cURL command. In this example, the object prefix is `photos/` and the file being uploaded is called `flower.jpg`.

This example uses the **swift-form-signature** script to compute the `expires` and `signature` values.

```
$ bin/swift-form-signature /v1/my_account/container/photos/ https://example.
↪com/done.html 5373952000 1 200 MYKEY
Expires: 1390825338
Signature: 35129416ebda2f1a21b3c2b8939850dfc63d8f43
```

```
$ curl -i https://swift-cluster.example.com/v1/my_account/container/photos/ -
↪X POST \
       -F max_file_size=5373952000 -F max_file_count=1 -F expires=1390825338 \
       -F signature=35129416ebda2f1a21b3c2b8939850dfc63d8f43 \
       -F redirect=https://example.com/done.html \
       -F file=@flower.jpg
```

6.8 Use Content-Encoding metadata

When you create an object or update its metadata, you can optionally set the `Content-Encoding` metadata. This metadata enables you to indicate that the object content is compressed without losing the identity of the underlying media type (`Content-Type`) of the file, such as a video.

Example Content-Encoding header request: HTTP

This example assigns an attachment type to the `Content-Encoding` header that indicates how the file is downloaded:

```
PUT /<api version>/<account>/<container>/<object> HTTP/1.1
Host: storage.clouddrive.com
X-Auth-Token: eaaafd18-0fed-4b3a-81b4-663c99ec1cbb
Content-Type: video/mp4
Content-Encoding: gzip
```

6.9 Use the Content-Disposition metadata

To override the default behavior for a browser, use the `Content-Disposition` header to specify the override behavior and assign this header to an object. For example, this header might specify that the browser use a download program to save this file rather than show the file, which is the default.

Example Override browser default behavior request: HTTP

This example assigns an attachment type to the `Content-Disposition` header. This attachment type indicates that the file is to be downloaded as `goodbye.txt`:

```
# curl -i  $publicURL/marktwain/goodbye -X POST -H "X-Auth-Token: $token" -H
↪"Content-Length: 14" -H "Content-Type: application/octet-stream" -H
↪"Content-Disposition: attachment; filename=goodbye.txt"
```

```
HTTP/1.1 202 Accepted
Content-Length: 76
Content-Type: text/html; charset=UTF-8
X-Trans-Id: txa9b5e57d7f354d7ea9f57-0052e17e13
X-Openstack-Request-Id: txa9b5e57d7f354d7ea9f57-0052e17e13
Date: Thu, 23 Jan 2014 20:39:47 GMT

<html><h1>Accepted</h1><p>The request is accepted for processing.</p></html>
```

6.10 Pseudo-hierarchical folders and directories

Although you cannot nest directories in OpenStack Object Storage, you can simulate a hierarchical structure within a single container by adding forward slash characters (/) in the object name. To navigate the pseudo-directory structure, you can use the `delimiter` query parameter. This example shows you how to use pseudo-hierarchical folders and directories.

Note: In this example, the objects reside in a container called `backups`. Within that container, the objects are organized in a pseudo-directory called `photos`. The container name is not displayed in the example, but it is a part of the object URLs. For instance, the URL of the picture `me.jpg` is `https://swift.example.com/v1/CF_xer7_343/backups/photos/me.jpg`.

6.10.1 List pseudo-hierarchical folders request: HTTP

To display a list of all the objects in the storage container, use `GET` without a `delimiter` or `prefix`.

```
$ curl -X GET -i -H "X-Auth-Token: $token" \
 $publicurl/v1/AccountString/backups
```

The system returns status code 2xx (between 200 and 299, inclusive) and the requested list of the objects.

```
photos/animals/cats/persian.jpg
photos/animals/cats/siamese.jpg
photos/animals/dogs/corgi.jpg
photos/animals/dogs/poodle.jpg
photos/animals/dogs/terrier.jpg
photos/me.jpg
photos/plants/fern.jpg
photos/plants/rose.jpg
```

Use the delimiter parameter to limit the displayed results. To use `delimiter` with pseudo-directories, you must use the parameter slash (/).

```
$ curl -X GET -i -H "X-Auth-Token: $token" \
 $publicurl/v1/AccountString/backups?delimiter=/
```

The system returns status code 2xx (between 200 and 299, inclusive) and the requested matching objects. Because you use the slash, only the pseudo-directory `photos/` displays. The returned values from a slash `delimiter` query are not real objects. The value will refer to a real object if it does not end with a slash. The pseudo-directories have no content-type, rather, each pseudo-directory has its own `subdir` entry in the response of JSON and XML results. For example:

```
[
  {
    "subdir": "photos/"
  }
]
```

```
<?xml version="1.0" encoding="UTF-8"?>
<container name="backups">
  <subdir name="photos/">
    <name>photos/</name>
  </subdir>
</container>
```

Use the `prefix` and `delimiter` parameters to view the objects inside a pseudo-directory, including further nested pseudo-directories.

```
$ curl -X GET -i -H "X-Auth-Token: $token" \
 $publicurl/v1/AccountString/backups?prefix=photos/&delimiter=/
```

The system returns status code 2xx (between 200 and 299, inclusive) and the objects and pseudo-directories within the top level pseudo-directory.

```
photos/animals/
photos/me.jpg
photos/plants/
```

```
[
  {
    "subdir": "photos/animals/"
  },
  {
    "hash": "b249a153f8f38b51e92916bbc6ea57ad",
    "last_modified": "2015-12-03T17:31:28.187370",
    "bytes": 2906,
    "name": "photos/me.jpg",
    "content_type": "image/jpeg"
  },
  {
    "subdir": "photos/plants/"
  }
]
```

```
<?xml version="1.0" encoding="UTF-8"?>
<container name="backups">
  <subdir name="photos/animals/">
    <name>photos/animals/</name>
  </subdir>
  <object>
    <name>photos/me.jpg</name>
    <hash>b249a153f8f38b51e92916bbc6ea57ad</hash>
    <bytes>2906</bytes>
    <content_type>image/jpeg</content_type>
    <last_modified>2015-12-03T17:31:28.187370</last_modified>
  </object>
  <subdir name="photos/plants/">
    <name>photos/plants/</name>
```

(continues on next page)

(continued from previous page)

```
  </subdir>
</container>
```

You can create an unlimited number of nested pseudo-directories. To navigate through them, use a longer `prefix` parameter coupled with the `delimiter` parameter. In this sample output, there is a pseudo-directory called `dogs` within the pseudo-directory `animals`. To navigate directly to the files contained within `dogs`, enter the following command:

```
$ curl -X GET -i -H "X-Auth-Token: $token" \
 $publicurl/v1/AccountString/backups?prefix=photos/animals/dogs/&delimiter=/
```

The system returns status code 2xx (between 200 and 299, inclusive) and the objects and pseudo-directories within the nested pseudo-directory.

```
photos/animals/dogs/corgi.jpg
photos/animals/dogs/poodle.jpg
photos/animals/dogs/terrier.jpg
```

6.11 Page through large lists of containers or objects

If you have a large number of containers or objects, you can use the `marker`, `limit`, and `end_marker` parameters to control how many items are returned in a list and where the list starts or ends. If you want to page backwards you can use the `reverse` parameter.

- **marker** When you request a list of containers or objects, Object Storage returns a maximum of 10,000 names for each request. To get subsequent names, you must make another request with the `marker` parameter. Set the `marker` parameter to the name of the last item returned in the previous list. You must URL-encode the `marker` value before you send the HTTP request. Object Storage returns a maximum of 10,000 names starting after the last item returned.
- **limit** To return fewer than 10,000 names, use the `limit` parameter. If the number of names returned equals the specified `limit` (or 10,000 if you omit the `limit` parameter), you can assume there are more names to list. If the number of names in the list is exactly divisible by the `limit` value, the last request has no content.
- **end_marker** Limits the result set to names that are less than the `end_marker` parameter value. You must URL-encode the `end_marker` value before you send the HTTP request.
- **reverse** By default, listings are returned sorted by name, ascending. If you include the `reverse=true` query parameter, the listing will be returned sorted by name, descending.

6.11.1 To page through a large list of containers

Assume the following list of container names:

```
apples
bananas
kiwis
oranges
pears
```

1. Use a `limit` of two:

```
# curl -i $publicURL/?limit=2 -X GET -H "X-Auth-Token: $token"
```

```
apples
bananas
```

   Because two container names are returned, there are more names to list.

2. Make another request with a `marker` parameter set to the name of the last item returned:

```
# curl -i $publicURL/?limit=2&amp;marker=bananas -X GET -H \
  X-Auth-Token: $token"
```

```
kiwis
oranges
```

   Again, two items are returned, and there might be more.

3. Make another request with a `marker` of the last item returned:

```
# curl -i $publicURL/?limit=2&amp;marker=oranges -X GET -H \"
  X-Auth-Token: $token"
```

```
pears
```

   You receive a one-item response, which is fewer than the `limit` number of names. This indicates that this is the end of the list.

4. Use the `end_marker` parameter to limit the result set to object names that are less than the `end_marker` parameter value:

```
# curl -i $publicURL/?end_marker=oranges -X GET -H \"
 X-Auth-Token: $token"
```

```
apples
bananas
kiwis
```

   You receive a result set of all container names before the `end-marker` value.

5. Use the `reverse` parameter to work from the back of the list:

```
# curl -i $publicURL/?reverse=true -X GET -H \"
   X-Auth-Token: $token"
```

```
pears
oranges
kiwis
bananas
apples
```

6. You can also combine parameters:

```
# curl -i $publicURL/?reverse=true&end_marker=kiwis -X GET -H \"
   X-Auth-Token: $token"
```

```
pears
oranges
```

6.12 Serialized response formats

By default, the Object Storage API uses a `text/plain` response format. In addition, both JSON and XML data serialization response formats are supported.

To define the response format, use one of these methods:

Method	Description
format= `format` query parameter	Append this parameter to the URL for a `GET` request, where `format` is `json` or `xml`.
`Accept` request header	Include this header in the `GET` request. The valid header values are: **text/plain** Plain text response format. The default. **application/jsontext** JSON data serialization response format. **application/xml** XML data serialization response format. **text/xml** XML data serialization response format.

6.12.1 Exampleă1.ăJSON example with format query parameter

For example, this request uses the `format` query parameter to ask for a JSON response:

```
$ curl -i $publicURL?format=json -X GET -H "X-Auth-Token: $token"
```

```
HTTP/1.1 200 OK
Content-Length: 96
X-Account-Object-Count: 1
X-Timestamp: 1389453423.35964
X-Account-Meta-Subject: Literature
X-Account-Bytes-Used: 14
X-Account-Container-Count: 2
Content-Type: application/json; charset=utf-8
Accept-Ranges: bytes
X-Trans-Id: tx274a77a8975c4a66aeb24-0052d95365
Date: Fri, 17 Jan 2014 15:59:33 GMT
```

Object Storage lists container names with additional information in JSON format:

```
[
   {
      "count":0,
      "bytes":0,
      "name":"janeausten"
   },
   {
      "count":1,
      "bytes":14,
      "name":"marktwain"
   }
]
```

6.12.2 Exampleă2.ăXML example with Accept header

This request uses the `Accept` request header to ask for an XML response:

```
$ curl -i $publicURL -X GET -H "X-Auth-Token: $token" -H \
  "Accept: application/xml; charset=utf-8"
```

```
HTTP/1.1 200 OK
Content-Length: 263
X-Account-Object-Count: 3
X-Account-Meta-Book: MobyDick
X-Timestamp: 1389453423.35964
X-Account-Bytes-Used: 47
X-Account-Container-Count: 2
Content-Type: application/xml; charset=utf-8
Accept-Ranges: bytes
```

(continues on next page)

(continued from previous page)

```
X-Trans-Id: txf0b4c9727c3e491694019-0052e03420
Date: Wed, 22 Jan 2014 21:12:00 GMT
```

Object Storage lists container names with additional information in XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<account name="AUTH_73f0aa26640f4971864919d0eb0f0880">
    <container>
        <name>janeausten</name>
        <count>2</count>
        <bytes>33</bytes>
    </container>
    <container>
        <name>marktwain</name>
        <count>1</count>
        <bytes>14</bytes>
    </container>
</account>
```

The remainder of the examples in this guide use standard, non-serialized responses. However, all `GET` requests that perform list operations accept the `format` query parameter or `Accept` request header.

6.13 Create static website

To discover whether your Object Storage system supports this feature, see *Discoverability*. Alternatively, check with your service provider.

You can use your Object Storage account to create a static website. This static website is created with Static Web middleware and serves container data with a specified index file, error file resolution, and optional file listings. This mode is normally active only for anonymous requests, which provide no authentication token. To use it with authenticated requests, set the header `X-Web-Mode` to `TRUE` on the request.

The Static Web filter must be added to the pipeline in your `/etc/swift/proxy-server.conf` file below any authentication middleware. You must also add a Static Web middleware configuration section.

Your publicly readable containers are checked for two headers, `X-Container-Meta-Web-Index` and `X-Container-Meta-Web-Error`. The `X-Container-Meta-Web-Error` header is discussed below, in the section called *Set error pages for static website*.

Use `X-Container-Meta-Web-Index` to determine the index file (or default page served, such as `index.html`) for your website. When someone initially enters your site, the `index.html` file displays automatically. If you create sub-directories for your site by creating pseudo-directories in your container, the index page for each sub-directory is displayed by default. If your pseudo-directory does not have a file with the same name as your index file, visits to the sub-directory return a 404 error.

You also have the option of displaying a list of files in your pseudo-directory instead of a web page. To do this, set the `X-Container-Meta-Web-Listings` header to `TRUE`. You may add styles to your file listing by setting `X-Container-Meta-Web-Listings-CSS` to a style sheet (for example, `lists.css`).

6.13.1 Static Web middleware through Object Storage

The following sections show how to use Static Web middleware through Object Storage.

Make container publicly readable

Make the container publicly readable. Once the container is publicly readable, you can access your objects directly, but you must set the index file to browse the main site URL and its sub-directories.

```
$ swift post -r '.r:*,.rlistings' container
```

Set site index file

Set the index file. In this case, `index.html` is the default file displayed when the site appears.

```
$ swift post -m 'web-index:index.html' container
```

Enable file listing

Turn on file listing. If you do not set the index file, the URL displays a list of the objects in the container. Instructions on styling the list with a CSS follow.

```
$ swift post -m 'web-listings: true' container
```

Enable CSS for file listing

Style the file listing using a CSS.

```
$ swift post -m 'web-listings-css:listings.css' container
```

Set error pages for static website

You can create and set custom error pages for visitors to your website; currently, only 401 (Unauthorized) and 404 (Not Found) errors are supported. To do this, set the metadata header, `X-Container-Meta-Web-Error`.

Error pages are served with the status code pre-pended to the name of the error page you set. For instance, if you set `X-Container-Meta-Web-Error` to `error.html`, 401 errors will display the page `401error.html`. Similarly, 404 errors will display `404error.html`. You must have both of these pages created in your container when you set the `X-Container-Meta-Web-Error` metadata, or your site will display generic error pages.

You only have to set the `X-Container-Meta-Web-Error` metadata once for your entire static website.

Set error pages for static website request

```
$ swift post -m 'web-error:error.html' container
```

Any 2nn response indicates success.

6.14 Object expiration

You can schedule Object Storage (swift) objects to expire by setting the `X-Delete-At` or `X-Delete-After` header. Once the object is deleted, swift will no longer serve the object and it will be deleted from the cluster shortly thereafter.

- Set an object to expire at an absolute time (in Unix time). You can get the current Unix time by running `date +'%s'`.

  ```
  $ swift post CONTAINER OBJECT_FILENAME -H "X-Delete-At:UNIX_TIME"
  ```

  Verify the `X-Delete-At` header has posted to the object:

  ```
  $ swift stat CONTAINER OBJECT_FILENAME
  ```

- Set an object to expire after a relative amount of time (in seconds):

  ```
  $ swift post CONTAINER OBJECT_FILENAME -H "X-Delete-After:SECONDS"
  ```

  The `X-Delete-After` header will be converted to `X-Delete-At`. Verify the `X-Delete-At` header has posted to the object:

  ```
  $ swift stat CONTAINER OBJECT_FILENAME
  ```

  If you no longer want to expire the object, you can remove the `X-Delete-At` header:

  ```
  $ swift post CONTAINER OBJECT_FILENAME -H "X-Remove-Delete-At:"
  ```

Note: In order for object expiration to work properly, the `swift-object-expirer` daemon will need access to all backend servers in the cluster. The daemon does not need access to the proxy-server or public network.

6.15 Bulk delete

To discover whether your Object Storage system supports this feature, see *Discoverability*. Alternatively, check with your service provider.

With bulk delete, you can delete up to 10,000 objects or containers (configurable) in one request.

6.15.1 Bulk delete request

To perform a bulk delete operation, add the `bulk-delete` query parameter to the path of a `POST` or `DELETE` operation.

Note: The `DELETE` operation is supported for backwards compatibility.

The path is the account, such as `/v1/12345678912345`, that contains the objects and containers.

In the request body of the `POST` or `DELETE` operation, list the objects or containers to be deleted. Separate each name with a newline character. You can include a maximum of 10,000 items (configurable) in the list.

In addition, you must:

- UTF-8-encode and then URL-encode the names.
- To indicate an object, specify the container and object name as: `CONTAINER_NAME/OBJECT_NAME`.
- To indicate a container, specify the container name as: `CONTAINER_NAME`. Make sure that the container is empty. If it contains objects, Object Storage cannot delete the container.
- Set the `Content-Type` request header to `text/plain`.

6.15.2 Bulk delete response

When Object Storage processes the request, it performs multiple sub-operations. Even if all sub-operations fail, the operation returns a 200 status. The bulk operation returns a response body that contains details that indicate which sub-operations have succeeded and failed. Some sub-operations might succeed while others fail. Examine the response body to determine the results of each delete sub-operation.

You can set the `Accept` request header to one of the following values to define the response format:

`text/plain` Formats response as plain text. If you omit the `Accept` header, `text/plain` is the default.

`application/json` Formats response as JSON.

`application/xml` or `text/xml` Formats response as XML.

The response body contains the following information:

- The number of files actually deleted.
- The number of not found objects.
- Errors. A list of object names and associated error statuses for the objects that failed to delete. The format depends on the value that you set in the `Accept` header.

The following bulk delete response is in `application/xml` format. In this example, the `mycontainer` container is not empty, so it cannot be deleted.

```
<delete>
    <number_deleted>2</number_deleted>
    <number_not_found>4</number_not_found>
    <errors>
```

(continues on next page)

(continued from previous page)

```
        <object>
            <name>/v1/12345678912345/mycontainer</name>
            <status>409 Conflict</status>
        </object>
    </errors>
</delete>
```

CHAPTER

SEVEN

S3 COMPATIBILITY INFO

7.1 S3/Swift REST API Comparison Matrix

7.1.1 General compatibility statement

S3 is a product from Amazon, and as such, it includes features that are outside the scope of Swift itself. For example, Swift doesnt have anything to do with billing, whereas S3 buckets can be tied to Amazons billing system. Similarly, log delivery is a service outside of Swift. Its entirely possible for a Swift deployment to provide that functionality, but it is not part of Swift itself. Likewise, a Swift deployment can provide similar geographic availability as S3, but this is tied to the deployers willingness to build the infrastructure and support systems to do so.

7.1.2 Amazon S3 operations

S3 REST API method	Category	Swift S3 API
GET Object	Core-API	Yes
HEAD Object	Core-API	Yes
PUT Object	Core-API	Yes
PUT Object Copy	Core-API	Yes
DELETE Object	Core-API	Yes
Initiate Multipart Upload	Core-API	Yes
Upload Part	Core-API	Yes
Upload Part Copy	Core-API	Yes
Complete Multipart Upload	Core-API	Yes
Abort Multipart Upload	Core-API	Yes
List Parts	Core-API	Yes
GET Object ACL	Core-API	Yes
PUT Object ACL	Core-API	Yes
PUT Bucket	Core-API	Yes
GET Bucket List Objects	Core-API	Yes
HEAD Bucket	Core-API	Yes
DELETE Bucket	Core-API	Yes
List Multipart Uploads	Core-API	Yes
GET Bucket acl	Core-API	Yes
PUT Bucket acl	Core-API	Yes
Versioning	Versioning	Yes

continues on next page

Table 1 – continued from previous page

S3 REST API method	Category	Swift S3 API
Bucket notification	Notifications	No
Bucket Lifecycle[1][2][3][4][5][6]	Bucket Lifecycle	No
Bucket policy	Advanced ACLs	No
Public website[7][8][9][10]	Public Website	No
Billing[11][12]	Billing	No
GET Bucket location	Advanced Feature	Yes
Delete Multiple Objects	Advanced Feature	Yes
Object tagging	Advanced Feature	No
GET Object torrent	Advanced Feature	No
Bucket inventory	Advanced Feature	No
GET Bucket service	Advanced Feature	No
Bucket accelerate	CDN Integration	No

[1] POST restore
[2] Bucket lifecycle
[3] Bucket logging
[4] Bucket analytics
[5] Bucket metrics
[6] Bucket replication
[7] OPTIONS object
[8] Object POST from HTML form
[9] Bucket public website
[10] Bucket CORS
[11] Request payment
[12] Bucket tagging

CHAPTER

EIGHT

OPENSTACK END USER GUIDE

The OpenStack End User Guide has additional information on using Swift. See the Manage objects and containers section.

CHAPTER

NINE

SOURCE DOCUMENTATION

9.1 Partitioned Consistent Hash Ring

9.1.1 Ring

class swift.common.ring.ring.**Ring**(*serialized_path*, *reload_time=None*, *ring_name=None*, *validation_hook=<function Ring.<lambda>*)

Bases: object

Partitioned consistent hashing ring.

Parameters

- **serialized_path** path to serialized RingData instance
- **reload_time** time interval in seconds to check for a ring change
- **ring_name** ring name string (basically specified from policy)
- **validation_hook** hook point to validate ring configuration ontime

Raises *RingLoadError* if the loaded ring data violates its constraint

property assigned_device_count

Number of devices with assignments in the ring.

property device_count

Number of devices in the ring.

property devs

devices in the ring

get_more_nodes(*part*)

Generator to get extra nodes for a partition for hinted handoff.

The handoff nodes will try to be in zones other than the primary zones, will take into account the device weights, and will usually keep the same sequences of handoffs even with ring changes.

Parameters **part** partition to get handoff nodes for

Returns generator of node dicts

See *get_nodes()* for a description of the node dicts.

get_nodes(*account*, *container=None*, *obj=None*)

Get the partition and nodes for an account/container/object. If a node is responsible for more than one replica, it will only appear in the output once.

Parameters

- **account** account name
- **container** container name
- **obj** object name

Returns a tuple of (partition, list of node dicts)

Each node dict will have at least the following keys:

id	unique integer identifier amongst devices
index	offset into the primary node list for the partition
weight	a float of the relative weight of this device as compared to others; this indicates how many partitions the builder will try to assign to this device
zone	integer indicating which zone the device is in; a given partition will not be assigned to multiple devices within the same zone
ip	the ip address of the device
port	the tcp port of the device
device	the devices name on disk (sdb1, for example)
meta	general use extra field; for example: the online date, the hardware description

get_part(*account*, *container=None*, *obj=None*)

Get the partition for an account/container/object.

Parameters

- **account** account name
- **container** container name
- **obj** object name

Returns the partition number

get_part_nodes(*part*)

Get the nodes that are responsible for the partition. If one node is responsible for more than one replica of the same partition, it will only appear in the output once.

Parameters **part** partition to get nodes for

Returns list of node dicts

See *get_nodes()* for a description of the node dicts.

has_changed()

Check to see if the ring on disk is different than the current one in memory.

Returns True if the ring on disk has changed, False otherwise

property md5

property next_part_power

property part_power

property partition_count

Number of partitions in the ring.

property raw_size

property replica_count

Number of replicas (full or partial) used in the ring.

property size

property version

property weighted_device_count

Number of devices with weight in the ring.

class swift.common.ring.ring.**RingData**(*replica2part2dev_id*, *devs*, *part_shift*, *next_part_power=None*, *version=None*)

Bases: object

Partitioned consistent hashing ring data (used for serialization).

classmethod deserialize_v1(*gz_file*, *metadata_only=False*)

Deserialize a v1 ring file into a dictionary with *devs*, *part_shift*, and *replica2part2dev_id* keys.

If the optional kwarg *metadata_only* is True, then the *replica2part2dev_id* is not loaded and that key in the returned dictionary just has the value *[]*.

Parameters

- **gz_file** (*file*) An opened file-like object which has already consumed the 6 bytes of magic and version.
- **metadata_only** (*bool*) If True, only load *devs* and *part_shift*

Returns A dict containing *devs*, *part_shift*, and *replica2part2dev_id*

classmethod load(*filename*, *metadata_only=False*)

Load ring data from a file.

Parameters

- **filename** Path to a file serialized by the save() method.
- **metadata_only** (*bool*) If True, only load *devs* and *part_shift*.

Returns A RingData instance containing the loaded data.

property replica_count

Number of replicas (full or partial) used in the ring.

save(*filename*, *mtime=1300507380.0*)

Serialize this RingData instance to disk.

Parameters

- **filename** File into which this instance should be serialized.

- **mtime** time used to override mtime for gzip, default or None if the caller wants to include time

serialize_v1(*file_obj*)

to_dict()

class swift.common.ring.ring.**RingReader**(*filename*)

Bases: object

chunk_size = 65536

property close

property md5

read(*amount=- 1*)

readinto(*buffer*)

readline()

seek(*pos*, *ref=0*)

swift.common.ring.ring.**calc_replica_count**(*replica2part2dev_id*)

9.1.2 Ring Builder

class swift.common.ring.builder.**RingBuilder**(*part_power*, *replicas*, *min_part_hours*)

Bases: object

Used to build swift.common.ring.RingData instances to be written to disk and used with swift.common.ring.Ring instances. See bin/swift-ring-builder for example usage.

The instance variable devs_changed indicates if the device information has changed since the last balancing. This can be used by tools to know whether a rebalance request is an isolated request or due to added, changed, or removed devices.

Parameters

- **part_power** number of partitions = 2**part_power.
- **replicas** number of replicas for each partition
- **min_part_hours** minimum number of hours between partition changes

add_dev(*dev*)

Add a device to the ring. This device dict should have a minimum of the following keys:

id	unique integer identifier amongst devices. Defaults to the next id if the id key is not provided in the dict
weight	a float of the relative weight of this device as compared to others; this indicates how many partitions the builder will try to assign to this device
region	integer indicating which region the device is in
zone	integer indicating which zone the device is in; a given partition will not be assigned to multiple devices within the same (region, zone) pair if there is any alternative
ip	the ip address of the device
port	the tcp port of the device
device	the devices name on disk (sdb1, for example)
meta	general use extra field; for example: the online date, the hardware description

Note: This will not rebalance the ring immediately as you may want to make multiple changes for a single rebalance.

Parameters **dev** device dict

Returns id of device (not used in the tree anymore, but unknown users may depend on it)

cancel_increase_partition_power()

Cancels a ring partition power increasement.

This sets the next_part_power to the current part_power. Object replicators will still skip replication, and a cleanup is still required. Finally, a finish_increase_partition_power needs to be run.

Returns False if next_part_power was not set or is equal to current part_power, otherwise True.

change_min_part_hours(*min_part_hours*)

Changes the value used to decide if a given partition can be moved again. This restriction is to give the overall system enough time to settle a partition to its new location before moving it to yet another location. While no data would be lost if a partition is moved several times quickly, it could make that data unreachable for a short period of time.

This should be set to at least the average full partition replication time. Starting it at 24 hours and then lowering it to what the replicator reports as the longest partition cycle is best.

Parameters **min_part_hours** new value for min_part_hours

copy_from(*builder*)

Reinitializes this RingBuilder instance from data obtained from the builder dict given. Code example:

```
b = RingBuilder(1, 1, 1)  # Dummy values
b.copy_from(builder)
```

This is to restore a RingBuilder that has had its b.to_dict() previously saved.

debug()

Temporarily enables debug logging, useful in tests, e.g.:

```
with rb.debug():
    rb.rebalance()
```

property ever_rebalanced

finish_increase_partition_power()

Finish the partition power increase.

The hard links from the old object locations should be removed by now.

classmethod from_dict(*builder_data*)

get_balance()

Get the balance of the ring. The balance value is the highest percentage of the desired amount of partitions a given device wants. For instance, if the worst device wants (based on its weight relative to the sum of all the devices weights) 123 partitions and it has 124 partitions, the balance value would be 0.83 (1 extra / 123 wanted * 100 for percentage).

Returns balance of the ring

get_part_devices(*part*)

Get the devices that are responsible for the partition, filtering out duplicates.

Parameters **part** partition to get devices for

Returns list of device dicts

get_required_overload(*weighted=None*, *wanted=None*)

Returns the minimum overload value required to make the ring maximally dispersed.

The required overload is the largest percentage change of any single device from its weighted replicanth to its wanted replicanth (note: under weighted devices have a negative percentage change) to archive dispersion - that is to say a single device that must be overloaded by 5% is worse than 5 devices in a single tier overloaded by 1%.

get_ring()

Get the ring, or more specifically, the swift.common.ring.RingData. This ring data is the minimum required for use of the ring. The ring builder itself keeps additional data such as when partitions were last moved.

property id

increase_partition_power()

Increases ring partition power by one.

Devices will be assigned to partitions like this:

OLD: 0, 3, 7, 5, 2, 1, NEW: 0, 0, 3, 3, 7, 7, 5, 5, 2, 2, 1, 1,

Returns False if next_part_power was not set or is equal to current part_power, None if something went wrong, otherwise True.

classmethod load(*builder_file*, *open=<built-in function open>*, ***kwargs*)

Obtain RingBuilder instance of the provided builder file

Parameters **builder_file** path to builder file to load

Returns RingBuilder instance

property min_part_seconds_left

Get the total seconds until a rebalance can be performed

property part_shift

prepare_increase_partition_power()

Prepares a ring for partition power increase.

This makes it possible to compute the future location of any object based on the next partition power.

In this phase object servers should create hard links when finalizing a write to the new location as well. A relinker will be run after restarting object-servers, creating hard links to all existing objects in their future location.

Returns False if next_part_power was not set, otherwise True.

pretend_min_part_hours_passed()

Override min_part_hours by marking all partitions as having been moved 255 hours ago and last move epoch to the beginning of time. This can be used to force a full rebalance on the next call to rebalance.

rebalance(*seed=None*)

Rebalance the ring.

This is the main work function of the builder, as it will assign and reassign partitions to devices in the ring based on weights, distinct zones, recent reassignments, etc.

The process doesnt always perfectly assign partitions (thatd take a lot more analysis and therefore a lot more time I had code that did that before). Because of this, it keeps rebalancing until the device skew (number of partitions a device wants compared to what it has) gets below 1% or doesnt change by more than 1% (only happens with a ring that cant be balanced no matter what).

Parameters **seed** a value for the random seed (optional)

Returns (number_of_partitions_altered, resulting_balance, number_of_removed_devices)

remove_dev(*dev_id*)

Remove a device from the ring.

Note: This will not rebalance the ring immediately as you may want to make multiple changes for a single rebalance.

Parameters **dev_id** device id

save(*builder_file*)

Serialize this RingBuilder instance to disk.

Parameters **builder_file** path to builder file to save

search_devs(*search_values*)

Search devices by parameters.

Parameters **search_values** a dictionary with search values to filter devices, supported parameters are id, region, zone, ip, port, replication_ip, replication_port, device, weight, meta

Returns list of device dicts

set_dev_region(*dev_id*, *region*)

Set the region of a device. This should be called rather than just altering the region key in the device dict directly, as the builder will need to rebuild some internal state to reflect the change.

Note: This will not rebalance the ring immediately as you may want to make multiple changes for a single rebalance.

Parameters

- **dev_id** device id
- **region** new region for device

set_dev_weight(*dev_id*, *weight*)

Set the weight of a device. This should be called rather than just altering the weight key in the device dict directly, as the builder will need to rebuild some internal state to reflect the change.

Note: This will not rebalance the ring immediately as you may want to make multiple changes for a single rebalance.

Parameters

- **dev_id** device id
- **weight** new weight for device

set_dev_zone(*dev_id*, *zone*)

Set the zone of a device. This should be called rather than just altering the zone key in the device dict directly, as the builder will need to rebuild some internal state to reflect the change.

Note: This will not rebalance the ring immediately as you may want to make multiple changes for a single rebalance.

Parameters

- **dev_id** device id
- **zone** new zone for device

set_overload(*overload*)

set_replicas(*new_replica_count*)

Changes the number of replicas in this ring.

If the new replica count is sufficiently different that self._replica2part2dev will change size, sets self.devs_changed. This is so tools like bin/swift-ring-builder can know to write out the new ring rather than bailing out due to lack of balance change.

to_dict()

Returns a dict that can be used later with copy_from to restore a RingBuilder. swift-ring-builder uses this to pickle.dump the dict to a file and later load that dict into copy_from.

validate(*stats=False*)

Validate the ring.

This is a safety function to try to catch any bugs in the building process. It ensures partitions have been assigned to real devices, arent doubly assigned, etc. It can also optionally check the even distribution of partitions across devices.

Parameters **stats** if True, check distribution of partitions across devices

Returns if stats is True, a tuple of (device_usage, worst_stat), else (None, None). device_usage[dev_id] will equal the number of partitions assigned to that device. worst_stat will equal the number of partitions the worst device is skewed from the number it should have.

Raises *RingValidationError* problem was found with the ring.

weight_of_one_part()

Returns the weight of each partition as calculated from the total weight of all the devices.

exception swift.common.ring.builder.**RingValidationWarning**

Bases: Warning

9.1.3 Composite Ring Builder

A standard ring built using the *ring-builder* will attempt to randomly disperse replicas or erasure-coded fragments across failure domains, but does not provide any guarantees such as placing at least one replica of every partition into each region. Composite rings are intended to provide operators with greater control over the dispersion of object replicas or fragments across a cluster, in particular when there is a desire to have strict guarantees that some replicas or fragments are placed in certain failure domains. This is particularly important for policies with duplicated erasure-coded fragments.

A composite ring comprises two or more component rings that are combined to form a single ring with a replica count equal to the sum of replica counts from the component rings. The component rings are built independently, using distinct devices in distinct regions, which means that the dispersion of replicas between the components can be guaranteed. The `composite_builder` utilities may then be used to combine components into a composite ring.

For example, consider a normal ring `ring0` with replica count of 4 and devices in two regions `r1` and `r2`. Despite the best efforts of the ring-builder, it is possible for there to be three replicas of a particular partition placed in one region and only one replica placed in the other region. For example:

```
part_n -> r1z1h110/sdb r1z2h12/sdb r1z3h13/sdb r2z1h21/sdb
```

Now consider two normal rings each with replica count of 2: `ring1` has devices in only `r1`; `ring2` has devices in only `r2`. When these rings are combined into a composite ring then every partition is guaranteed to be mapped to two devices in each of `r1` and `r2`, for example:

```
part_n -> r1z1h10/sdb r1z2h20/sdb  r2z1h21/sdb r2z2h22/sdb
          |_____________________|  |_____________________|
                     |                        |
                   ring1                    ring2
```

The dispersion of partition replicas across failure domains within each of the two component rings may change as they are modified and rebalanced, but the dispersion of replicas between the two regions is guaranteed by the use of a composite ring.

For rings to be formed into a composite they must satisfy the following requirements:

- All component rings must have the same part power (and therefore number of partitions)
- All component rings must have an integer replica count
- Each region may only be used in one component ring
- Each device may only be used in one component ring

Under the hood, the composite ring has a `_replica2part2dev_id` table that is the union of the tables from the component rings. Whenever the component rings are rebalanced, the composite ring must be rebuilt. There is no dynamic rebuilding of the composite ring.

Note: The order in which component rings are combined into a composite ring is very significant because it determines the order in which the Ring.get_part_nodes() method will provide primary nodes for the composite ring and consequently the node indexes assigned to the primary nodes. For an erasure-coded policy, inadvertent changes to the primary node indexes could result in large amounts of data movement due to fragments being moved to their new correct primary.

The `id` of each component RingBuilder is therefore stored in metadata of the composite and used to check for the component ordering when the same composite ring is re-composed. RingBuilder `id`s are normally assigned when a RingBuilder instance is first saved. Older RingBuilder instances loaded from file may not have an `id` assigned and will need to be saved before they can be used as components of a composite ring. This can be achieved by, for example:

```
swift-ring-builder <builder-file> rebalance --force
```

---

class swift.common.ring.composite_builder.**CompositeRingBuilder**(*builder_files=None*)

Bases: `object`

Provides facility to create, persist, load, rebalance and update composite rings, for example:

```
# create a CompositeRingBuilder instance with a list of
# component builder files
crb = CompositeRingBuilder(["region1.builder", "region2.builder"])

# perform a cooperative rebalance of the component builders
```

(continues on next page)

(continued from previous page)

```
crb.rebalance()

# call compose which will make a new RingData instance
ring_data = crb.compose()

# save the composite ring file
ring_data.save("composite_ring.gz")

# save the composite metadata file
crb.save("composite_builder.composite")

# load the persisted composite metadata file
crb = CompositeRingBuilder.load("composite_builder.composite")

# compose (optionally update the paths to the component builder files)
crb.compose(["/path/to/region1.builder", "/path/to/region2.builder"])
```

Composite ring metadata is persisted to file in JSON format. The metadata has the structure shown below (using example values):

```
{
  "version": 4,
  "components": [
    {
      "version": 3,
      "id": "8e56f3b692d43d9a666440a3d945a03a",
      "replicas": 1
    },
    {
      "version": 5,
      "id": "96085923c2b644999dbfd74664f4301b",
      "replicas": 1
    }
  ]
  "component_builder_files": {
      "8e56f3b692d43d9a666440a3d945a03a": "/etc/swift/region1.builder",
      "96085923c2b644999dbfd74664f4301b": "/etc/swift/region2.builder",
  }
  "serialization_version": 1,
  "saved_path": "/etc/swift/multi-ring-1.composite",
}
```

version is an integer representing the current version of the composite ring, which increments each time the ring is successfully (re)composed.

components is a list of dicts, each of which describes relevant properties of a component ring

component_builder_files is a dict that maps component ring builder ids to the file from which that component ring builder was loaded.

serialization_version is an integer constant.

saved_path is the path to which the metadata was written.

Params builder_files a list of paths to builder files that will be used as components of the composite ring.

`can_part_move`(*part*)

Check with all component builders that it is ok to move a partition.

Parameters `part` The partition to check.

Returns True if all component builders agree that the partition can be moved, False otherwise.

`compose`(*builder_files=None*, *force=False*, *require_modified=False*)

Builds a composite ring using component ring builders loaded from a list of builder files and updates composite ring metadata.

If a list of component ring builder files is given then that will be used to load component ring builders. Otherwise, component ring builders will be loaded using the list of builder files that was set when the instance was constructed.

In either case, if metadata for an existing composite ring has been loaded then the component ring builders are verified for consistency with the existing composition of builders, unless the optional `force` flag if set True.

Parameters

- `builder_files` Optional list of paths to ring builder files that will be used to load the component ring builders. Typically the list of component builder files will have been set when the instance was constructed, for example when using the load() class method. However, this parameter may be used if the component builder file paths have moved, or, in conjunction with the `force` parameter, if a new list of component builders is to be used.
- `force` if True then do not verify given builders are consistent with any existing composite ring (default is False).
- `require_modified` if True and `force` is False, then verify that at least one of the given builders has been modified since the composite ring was last built (default is False).

Returns An instance of *swift.common.ring.ring.RingData*

Raises ValueError if the component ring builders are not suitable for composing with each other, or are inconsistent with any existing composite ring, or if require_modified is True and there has been no change with respect to the existing ring.

`classmethod load`(*path_to_file*)

Load composite ring metadata.

Parameters `path_to_file` Absolute path to a composite ring JSON file.

Returns an instance of *CompositeRingBuilder*

Raises

- `IOError` if there is a problem opening the file
- `ValueError` if the file does not contain valid composite ring metadata

load_components(*builder_files=None*, *force=False*, *require_modified=False*)

Loads component ring builders from builder files. Previously loaded component ring builders will discarded and reloaded.

If a list of component ring builder files is given then that will be used to load component ring builders. Otherwise, component ring builders will be loaded using the list of builder files that was set when the instance was constructed.

In either case, if metadata for an existing composite ring has been loaded then the component ring builders are verified for consistency with the existing composition of builders, unless the optional `force` flag if set True.

Parameters

- **builder_files** Optional list of paths to ring builder files that will be used to load the component ring builders. Typically the list of component builder files will have been set when the instance was constructed, for example when using the load() class method. However, this parameter may be used if the component builder file paths have moved, or, in conjunction with the `force` parameter, if a new list of component builders is to be used.
- **force** if True then do not verify given builders are consistent with any existing composite ring (default is False).
- **require_modified** if True and `force` is False, then verify that at least one of the given builders has been modified since the composite ring was last built (default is False).

Returns A tuple of (builder files, loaded builders)

Raises ValueError if the component ring builders are not suitable for composing with each other, or are inconsistent with any existing composite ring, or if require_modified is True and there has been no change with respect to the existing ring.

rebalance()

Cooperatively rebalances all component ring builders.

This method does not change the state of the composite ring; a subsequent call to *compose()* is required to generate updated composite `RingData`.

Returns

A list of dicts, one per component builder, each having the following keys:

- builder_file maps to the component builder file;
- builder maps to the corresponding instance of *swift.common.ring.builder.RingBuilder*;
- result maps to the results of the rebalance of that component i.e. a tuple of: *(number_of_partitions_altered, resulting_balance, number_of_removed_devices)*

The list has the same order as components in the composite ring.

Raises *RingBuilderError* if there is an error while rebalancing any component builder.

save(*path_to_file*)

Save composite ring metadata to given file. See *CompositeRingBuilder* for details of the persisted metadata format.

Parameters **path_to_file** Absolute path to a composite ring file

Raises **ValueError** if no composite ring has been built yet with this instance

to_dict()

Transform the composite ring attributes to a dict. See *CompositeRingBuilder* for details of the persisted metadata format.

Returns a composite ring metadata dict

update_last_part_moves()

Updates the record of how many hours ago each partition was moved in all component builders.

class swift.common.ring.composite_builder.**CooperativeRingBuilder**(*part_power*, *replicas*, *min_part_hours*, *parent_builder*)

Bases: *swift.common.ring.builder.RingBuilder*

A subclass of RingBuilder that participates in cooperative rebalance.

During rebalance this subclass will consult with its *parent_builder* before moving a partition. The *parent_builder* may in turn check with co-builders (including this instance) to verify that none have moved that partition in the last *min_part_hours*.

Parameters

- **part_power** number of partitions = 2**part_power.
- **replicas** number of replicas for each partition.
- **min_part_hours** minimum number of hours between partition changes.
- **parent_builder** an instance of *CompositeRingBuilder*.

can_part_move(*part*)

Check that in the context of this builder alone it is ok to move a partition.

Parameters **part** The partition to check.

Returns True if the partition can be moved, False otherwise.

update_last_part_moves()

Updates the record of how many hours ago each partition was moved in in this builder.

swift.common.ring.composite_builder.**check_against_existing**(*old_composite_meta*, *new_composite_meta*)

Check that the given builders and their order are the same as that used to build an existing composite ring. Return True if any of the given builders has been modified with respect to its state when the given component_meta was created.

Parameters

- **old_composite_meta** a dict of the form returned by _make_composite_meta()

- **new_composite_meta** a dict of the form returned by _make_composite_meta()

Returns True if any of the components has been modified, False otherwise.

Raises **Value Error** if proposed new components do not match any existing components.

swift.common.ring.composite_builder.**check_builder_ids**(*builders*)

Check that all builders in the given list have ids assigned and that no id appears more than once in the list.

Parameters **builders** a list instances of *swift.common.ring.builder.RingBuilder*

Raises ValueError if any builder id is missing or repeated

swift.common.ring.composite_builder.**check_for_dev_uniqueness**(*builders*)

Check that no device appears in more than one of the given list of builders.

Parameters **builders** a list of *swift.common.ring.builder.RingBuilder* instances

Raises **ValueError** if the same device is found in more than one builder

swift.common.ring.composite_builder.**check_same_builder**(*old_component*, *new_component*)

Check that the given new_component metadata describes the same builder as the given old_component metadata. The new_component builder does not necessarily need to be in the same state as when the old_component metadata was created to satisfy this check e.g. it may have changed devs and been rebalanced.

Parameters

- **old_component** a dict of metadata describing a component builder
- **new_component** a dict of metadata describing a component builder

Raises **ValueError** if the new_component is not the same as that described by the old_component

swift.common.ring.composite_builder.**compose_rings**(*builders*)

Given a list of component ring builders, perform validation on the list of builders and return a composite RingData instance.

Parameters **builders** a list of *swift.common.ring.builder.RingBuilder* instances

Returns a new RingData instance built from the component builders

Raises **ValueError** if the builders are invalid with respect to each other

swift.common.ring.composite_builder.**is_builder_newer**(*old_component*, *new_component*)

Return True if the given builder has been modified with respect to its state when the given component_meta was created.

Parameters

- **old_component** a dict of metadata describing a component ring

- **new_component** a dict of metadata describing a component ring

Returns True if the builder has been modified, False otherwise.

Raises **ValueError** if the version of the new_component is older than the version of the existing component.

swift.common.ring.composite_builder.**pre_validate_all_builders**(*builders*)

Pre-validation for all component ring builders that are to be included in the composite ring. Checks that all component rings are valid with respect to each other.

Parameters **builders** a list of *swift.common.ring.builder.RingBuilder* instances

Raises **ValueError** if the builders are invalid with respect to each other

9.2 Proxy

9.2.1 Proxy Controllers

Base

class swift.proxy.controllers.base.**ByteCountEnforcer**(*file_like*, *nbytes*)

Bases: object

Enforces that successive calls to file_like.read() give at least <nbytes> bytes before exhaustion.

If file_like fails to do so, ShortReadError is raised.

If more than <nbytes> bytes are read, we dont care.

read(*amt=None*)

class swift.proxy.controllers.base.**Controller**(*app*)

Bases: object

Base WSGI controller class for the proxy

GET(*req*)

Handler for HTTP GET requests.

Parameters **req** The client request

Returns the response to the client

GETorHEAD_base(*req*, *server_type*, *node_iter*, *partition*, *path*, *concurrency=1*, *policy=None*, *client_chunk_size=None*)

Base handler for HTTP GET or HEAD requests.

Parameters

- **req** swob.Request object
- **server_type** server type used in logging
- **node_iter** an iterator to obtain nodes from
- **partition** partition

- **path** path for the request
- **concurrency** number of requests to run concurrently
- **policy** the policy instance, or None if Account or Container
- **client_chunk_size** chunk size for response body iterator

Returns swob.Response object

HEAD(*req*)

Handler for HTTP HEAD requests.

Parameters **req** The client request

Returns the response to the client

OPTIONS(*req*)

Base handler for OPTIONS requests

Parameters **req** swob.Request object

Returns swob.Response object

account_info(*account*, *req*)

Get account information, and also verify that the account exists.

Parameters

- **account** native str name of the account to get the info for
- **req** callers HTTP request context object

Returns tuple of (account partition, account nodes, container_count) or (None, None, None) if it does not exist

property allowed_methods

autocreate_account(*req*, *account*)

Autocreate an account

Parameters

- **req** request leading to this autocreate
- **account** the unquoted account name

best_response(*req*, *statuses*, *reasons*, *bodies*, *server_type*, *etag=None*, *headers=None*, *overrides=None*, *quorum_size=None*)

Given a list of responses from several servers, choose the best to return to the API.

Parameters

- **req** swob.Request object
- **statuses** list of statuses returned
- **reasons** list of reasons for each status
- **bodies** bodies of each response
- **server_type** type of server the responses came from
- **etag** etag

- **headers** headers of each response
- **overrides** overrides to apply when lacking quorum
- **quorum_size** quorum size to use

Returns swob.Response object with the correct status, body, etc. set

container_info(*account*, *container*, *req*)

Get container information and thusly verify container existence. This will also verify account existence.

Parameters

- **account** native-str account name for the container
- **container** native-str container name to look up
- **req** callers HTTP request context object

Returns dict containing at least container partition (partition), container nodes (containers), container read acl (read_acl), container write acl (write_acl), and container sync key (sync_key). Values are set to None if the container does not exist.

generate_request_headers(*orig_req=None*, *additional=None*, *transfer=False*)

Create a list of headers to be used in backend requests

Parameters

- **orig_req** the original request sent by the client to the proxy
- **additional** additional headers to send to the backend
- **transfer** If True, transfer headers from original client request

Returns a dictionary of headers

get_name_length_limit()

have_quorum(*statuses*, *node_count*, *quorum=None*)

Given a list of statuses from several requests, determine if a quorum response can already be decided.

Parameters

- **statuses** list of statuses returned
- **node_count** number of nodes being queried (basically ring count)
- **quorum** number of statuses required for quorum

Returns True or False, depending on if quorum is established

is_origin_allowed(*cors_info*, *origin*)

Is the given Origin allowed to make requests to this resource

Parameters

- **cors_info** the resources CORS related metadata headers
- **origin** the origin making the request

Returns True or False

make_requests(*req*, *ring*, *part*, *method*, *path*, *headers*, *query_string=''*, *overrides=None*, *node_count=None*, *node_iterator=None*, *body=None*)

Sends an HTTP request to multiple nodes and aggregates the results. It attempts the primary nodes concurrently, then iterates over the handoff nodes as needed.

Parameters

- **req** a request sent by the client
- **ring** the ring used for finding backend servers
- **part** the partition number
- **method** the method to send to the backend
- **path** the path to send to the backend (full path ends up being /<$device>/<$part>/<$path>)
- **headers** a list of dicts, where each dict represents one backend request that should be made.
- **query_string** optional query string to send to the backend
- **overrides** optional return status override map used to override the returned status of a request.
- **node_count** optional number of nodes to send request to.
- **node_iterator** optional node iterator.

Returns a swob.Response object

pass_through_headers = []

property private_methods

server_type = 'Base'

transfer_headers(*src_headers*, *dst_headers*)

Transfer legal headers from an original client request to dictionary that will be used as headers by the backend request

Parameters

- **src_headers** A dictionary of the original client request headers
- **dst_headers** A dictionary of the backend request headers

class swift.proxy.controllers.base.**GetOrHeadHandler**(*app*, *req*, *server_type*, *node_iter*, *partition*, *path*, *backend_headers*, *concurrency=1*, *policy=None*, *client_chunk_size=None*, *newest=None*, *logger=None*)

Bases: object

fast_forward(*num_bytes*)

Will skip num_bytes into the current ranges.

Params num_bytes the number of bytes that have already been read on this request. This will change the Range header so that the next req will start where it left off.

Raises

- **HTTPRequestedRangeNotSatisfiable** if begin + num_bytes > end of range + 1
- *RangeAlreadyComplete* if begin + num_bytes == end of range + 1

get_working_response(*req*)

is_good_source(*src*)

Indicates whether or not the request made to the backend found what it was looking for.

Parameters **src** the response from the backend

Returns True if found, False if not

property last_headers

property last_status

learn_size_from_content_range(*start*, *end*, *length*)

If client_chunk_size is set, makes sure we yield things starting on chunk boundaries based on the Content-Range header in the response.

Sets our Range headers first byterange to the value learned from the Content-Range header in the response; if we were given a fully-specified range (e.g. bytes=123-456), this is a no-op.

If we were given a half-specified range (e.g. bytes=123- or bytes=-456), then this changes the Range header to a semantically-equivalent one *and* it lets us resume on a proper boundary instead of just in the middle of a piece somewhere.

pop_range()

Remove the first byterange from our Range header.

This is used after a byterange has been completely sent to the client; this way, should we need to resume the download from another object server, we do not re-fetch byteranges that the client already has.

If we have no Range header, this is a no-op.

class swift.proxy.controllers.base.**NodeIter**(*app*, *ring*, *partition*, *logger*, *node_iter=None*, *policy=None*)

Bases: object

Yields nodes for a ring partition, skipping over error limited nodes and stopping at the configurable number of nodes. If a node yielded subsequently gets error limited, an extra node will be yielded to take its place.

Note that if youre going to iterate over this concurrently from multiple greenthreads, youll want to use a swift.common.utils.GreenthreadSafeIterator to serialize access. Otherwise, you may get ValueErrors from concurrent access. (You also may not, depending on how logging is configured, the vagaries of socket IO and eventlet, and the phase of the moon.)

Parameters

- **app** a proxy app
- **ring** ring to get yield nodes from
- **partition** ring partition to yield nodes for

- **logger** a logger instance
- **node_iter** optional iterable of nodes to try. Useful if you want to filter or reorder the nodes.
- **policy** an instance of `BaseStoragePolicy`. This should be None for an account or container ring.

log_handoffs(*handoffs*)

Log handoff requests if handoff logging is enabled and the handoff was not expected.

We only log handoffs when weve pushed the handoff count further than we would normally have expected under normal circumstances, that is (request_node_count - num_primaries), when handoffs goes higher than that it means one of the primaries must have been skipped because of error limiting before we consumed all of our nodes_left.

next()

property primaries_left

set_node_provider(*callback*)

Install a callback function that will be used during a call to next() to get an alternate node instead of returning the next node from the iterator.

Parameters **callback** A no argument function that should return a node dict or None.

swift.proxy.controllers.base.**bytes_to_skip**(*record_size*, *range_start*)

Assume an object is composed of N records, where the first N-1 are all the same size and the last is at most that large, but may be smaller.

When a range request is made, it might start with a partial record. This must be discarded, lest the consumer get bad data. This is particularly true of suffix-byte-range requests, e.g. Range: bytes=-12345 where the size of the object is unknown at the time the request is made.

This function computes the number of bytes that must be discarded to ensure only whole records are yielded. Erasure-code decoding needs this.

This function could have been inlined, but it took enough tries to get right that some targeted unit tests were desirable, hence its extraction.

swift.proxy.controllers.base.**clear_info_cache**(*app*, *env*, *account*, *container=None*, *shard=None*)

Clear the cached info in both memcache and env

Parameters

- **app** the application object
- **env** the WSGI environment
- **account** the account name
- **container** the container name if clearing info for containers, or None
- **shard** the sharding state if clearing info for container shard ranges, or None

swift.proxy.controllers.base.**close_swift_conn**(*src*)

Force close the http connection to the backend.

Parameters **src** the response from the backend

swift.proxy.controllers.base.**cors_validation**(*func*)

Decorator to check if the request is a CORS request and if so, if its valid.

Parameters **func** function to check

swift.proxy.controllers.base.**delay_denial**(*func*)

Decorator to declare which methods should have any swift.authorize call delayed. This is so the method can load the Request object up with additional information that may be needed by the authorization system.

Parameters **func** function for which authorization will be delayed

swift.proxy.controllers.base.**get_account_info**(*env*, *app*, *swift_source=None*)

Get the info structure for an account, based on env and app. This is useful to middlewares.

Note: This call bypasses auth. Success does not imply that the request has authorization to the account.

Raises **ValueError** when path doesnt contain an account

swift.proxy.controllers.base.**get_cache_key**(*account*, *container=None*, *obj=None*, *shard=None*)

Get the keys for both memcache and env[swift.infocache] (cache_key) where info about accounts, containers, and objects is cached

Parameters

- **account** The name of the account
- **container** The name of the container (or None if account)
- **obj** The name of the object (or None if account or container)
- **shard** Sharding state for the container query; typically updating or listing (Requires account and container; cannot use with obj)

Returns a (native) string cache_key

swift.proxy.controllers.base.**get_container_info**(*env*, *app*, *swift_source=None*)

Get the info structure for a container, based on env and app. This is useful to middlewares.

Note: This call bypasses auth. Success does not imply that the request has authorization to the container.

---

swift.proxy.controllers.base.**get_info**(*app*, *env*, *account*, *container=None*, *swift_source=None*)

Get info about accounts or containers

Note: This call bypasses auth. Success does not imply that the request has authorization to the info.

Parameters

- **app** the application object
- **env** the environment used by the current request
- **account** The unquoted name of the account
- **container** The unquoted name of the container (or None if account)
- **swift_source** swift source logged for any subrequests made while retrieving the account or container info

Returns information about the specified entity in a dictionary. See get_account_info and get_container_info for details on whats in the dictionary.

swift.proxy.controllers.base.**get_object_info**(*env*, *app*, *path=None*, *swift_source=None*)

Get the info structure for an object, based on env and app. This is useful to middlewares.

Note: This call bypasses auth. Success does not imply that the request has authorization to the object.

---

swift.proxy.controllers.base.**headers_from_container_info**(*info*)

Construct a HeaderKeyDict from a container info dict.

Parameters **info** a dict of container metadata

Returns a HeaderKeyDict or None if info is None or any required headers could not be constructed

swift.proxy.controllers.base.**headers_to_account_info**(*headers*, *status_int=200*)

Construct a cacheable dict of account info based on response headers.

swift.proxy.controllers.base.**headers_to_container_info**(*headers*, *status_int=200*)

Construct a cacheable dict of container info based on response headers.

swift.proxy.controllers.base.**headers_to_object_info**(*headers*, *status_int=200*)

Construct a cacheable dict of object info based on response headers.

swift.proxy.controllers.base.**set_info_cache**(*app*, *env*, *account*, *container*, *resp*)

Cache info in both memcache and env.

Parameters

- **app** the application object
- **account** the unquoted account name
- **container** the unquoted container name or None
- **resp** the response received or None if info cache should be cleared

Returns the info that was placed into the cache, or None if the request status was not in (404, 410, 2xx).

swift.proxy.controllers.base.**set_object_info_cache**(*app*, *env*, *account*, *container*, *obj*, *resp*)

Cache object info in the WSGI environment, but not in memcache. Caching in memcache would lead to cache pressure and mass evictions due to the large number of objects in a typical Swift cluster. This is a per-request cache only.

Parameters

- **app** the application object
- **env** the environment used by the current request
- **account** the unquoted account name
- **container** the unquoted container name
- **obj** the unquoted object name
- **resp** a GET or HEAD response received from an object server, or None if info cache should be cleared

Returns the object info

swift.proxy.controllers.base.**source_key**(*resp*)

Provide the timestamp of the swift http response as a floating point value. Used as a sort key.

Parameters **resp** bufferedhttp response object

swift.proxy.controllers.base.**update_headers**(*response*, *headers*)

Helper function to update headers in the response.

Parameters

- **response** swob.Response object
- **headers** dictionary headers

Account

class swift.proxy.controllers.account.**AccountController**(*app*, *account_name*, ***kwargs*)

Bases: *swift.proxy.controllers.base.Controller*

WSGI controller for account requests

DELETE(*req*)

HTTP DELETE request handler.

GETorHEAD(*req*)

Handler for HTTP GET/HEAD requests.

POST(*req*)

HTTP POST request handler.

PUT(*req*)

HTTP PUT request handler.

add_acls_from_sys_metadata(*resp*)

server_type = 'Account'

Container

class swift.proxy.controllers.container.**ContainerController**(*app*, *account_name*, *container_name*, ***kwargs*)

Bases: *swift.proxy.controllers.base.Controller*

WSGI controller for container requests

DELETE(*req*)

HTTP DELETE request handler.

GET(*req*)

Handler for HTTP GET requests.

GETorHEAD(*req*)

Handler for HTTP GET/HEAD requests.

HEAD(*req*)

Handler for HTTP HEAD requests.

POST(*req*)

HTTP POST request handler.

PUT(*req*)

HTTP PUT request handler.

UPDATE(*req*)

HTTP UPDATE request handler.

Method to perform bulk operations on container DBs, similar to a merge_items REPLICATE request.

Not client facing; internal clients or middlewares must include `X-Backend-Allow-Method: UPDATE` header to access.

clean_acls(*req*)

pass_through_headers = ['x-container-read', 'x-container-write', 'x-container-sync-key', 'x-container-sync-to', 'x-versions-location']

server_type = 'Container'

Object

class swift.proxy.controllers.obj.**BaseObjectController**(*app*, *account_name*, *container_name*, *object_name*, ***kwargs*)

Bases: *swift.proxy.controllers.base.Controller*

Base WSGI controller for object requests.

DELETE(*req*)

HTTP DELETE request handler.

GET(*req*)

Handler for HTTP GET requests.

GETorHEAD(*req*)

Handle HTTP GET or HEAD requests.

HEAD(*req*)

Handler for HTTP HEAD requests.

POST(*req*)

HTTP POST request handler.

PUT(*req*)

HTTP PUT request handler.

iter_nodes_local_first(*ring*, *partition*, *policy=None*, *local_handoffs_first=False*)

Yields nodes for a ring partition.

If the write_affinity setting is non-empty, then this will yield N local nodes (as defined by the write_affinity setting) first, then the rest of the nodes as normal. It is a re-ordering of the nodes such that the local ones come first; no node is omitted. The effect is that the request will be serviced by local object servers first, but nonlocal ones will be employed if not enough local ones are available.

Parameters

- **ring** ring to get nodes from
- **partition** ring partition to yield nodes for
- **policy** optional, an instance of *BaseStoragePolicy*
- **local_handoffs_first** optional, if True prefer primaries and local handoff nodes first before looking elsewhere.

server_type = 'Object'

class swift.proxy.controllers.obj.**ECAppIter**(*path*, *policy*, *internal_parts_iters*, *range_specs*, *fa_length*, *obj_length*, *logger*)

Bases: object

WSGI iterable that decodes EC fragment archives (or portions thereof) into the original object (or portions thereof).

Parameters

- **path** objects path, sans v1 (e.g. /a/c/o)
- **policy** storage policy for this object
- **internal_parts_iters** list of the response-document-parts iterators for the backend GET responses. For an M+K erasure code, the caller must supply M such iterables.
- **range_specs** list of dictionaries describing the ranges requested by the client. Each dictionary contains the start and end of the clients requested byte range as well as the start and end of the EC segments containing that byte range.
- **fa_length** length of the fragment archive, in bytes, if the response is a 200. If its a 206, then this is ignored.

- **obj_length** length of the object, in bytes. Learned from the headers in the GET response from the object server.
- **logger** a logger

app_iter_range(*start*, *end*)

app_iter_ranges(*ranges*, *content_type*, *boundary*, *content_size*)

close()

kickoff(*req*, *resp*)

Start pulling data from the backends so that we can learn things like the real Content-Type that might only be in the multipart/byteranges response body. Update our response accordingly.

Also, this is the first point at which we can learn the MIME boundary that our response has in the headers. We grab that so we can also use it in the body.

Returns None

Raises *HTTPException* on error

next()

class swift.proxy.controllers.obj.**ECFragGetter**(*app*, *req*, *node_iter*, *partition*, *policy*, *path*, *backend_headers*, *header_provider*, *logger_thread_locals*, *logger*)

Bases: object

fast_forward(*num_bytes*)

Will skip num_bytes into the current ranges.

Params num_bytes the number of bytes that have already been read on this request. This will change the Range header so that the next req will start where it left off.

Raises

- **HTTPRequestedRangeNotSatisfiable** if begin + num_bytes > end of range + 1
- *RangeAlreadyComplete* if begin + num_bytes == end of range + 1

property last_headers

property last_status

learn_size_from_content_range(*start*, *end*, *length*)

If client_chunk_size is set, makes sure we yield things starting on chunk boundaries based on the Content-Range header in the response.

Sets our Range headers first byterange to the value learned from the Content-Range header in the response; if we were given a fully-specified range (e.g. bytes=123-456), this is a no-op.

If we were given a half-specified range (e.g. bytes=123- or bytes=-456), then this changes the Range header to a semantically-equivalent one *and* it lets us resume on a proper boundary instead of just in the middle of a piece somewhere.

pop_range()

Remove the first byterange from our Range header.

This is used after a byterange has been completely sent to the client; this way, should we need to resume the download from another object server, we do not re-fetch byteranges that the client already has.

If we have no Range header, this is a no-op.

response_parts_iter(*req*)

property source_and_node_iter

class swift.proxy.controllers.obj.**ECGetResponseBucket**(*policy*, *timestamp*)

Bases: object

A helper class to encapsulate the properties of buckets in which fragment getters and alternate nodes are collected.

add_alternate_nodes(*node*, *frag_indexes*)

add_response(*getter*, *parts_iter*)

Add another response to this bucket. Response buckets can be for fragments with the same timestamp, or for errors with the same status.

close_conns()

Close buckets responses; they wont be used for a client response.

property durable

get_responses()

Return a list of all useful sources. Where there are multiple sources associated with the same frag_index then only one is included.

Returns a list of sources, each source being a tuple of form (ECFragGetter, iter)

set_durable()

property shortfall

The number of additional responses needed to complete this bucket; typically (ndata - resp_count).

If the bucket has no durable responses, shortfall is extended out to replica count to ensure the proxy makes additional primary requests.

property shortfall_with_alts

class swift.proxy.controllers.obj.**ECGetResponseCollection**(*policy*)

Bases: object

Manages all successful EC GET responses gathered by ECFragGetters.

A response comprises a tuple of (<getter instance>, <parts iterator>). All responses having the same data timestamp are placed in an ECGetResponseBucket for that timestamp. The buckets are stored in the buckets dict which maps timestamp-> bucket.

This class encapsulates logic for selecting the best bucket from the collection, and for choosing alternate nodes.

add_bad_resp(*get*, *parts_iter*)

add_good_response(*get*, *parts_iter*)

add_response(*get*, *parts_iter*)

Add a response to the collection.

Parameters

- **get** An instance of *ECFragGetter*
- **parts_iter** An iterator over response body parts

Raises **ValueError** if the response etag or status code values do not match any values previously received for the same timestamp

property best_bucket

Return the best bucket in the collection.

The best bucket is the newest timestamp with sufficient getters, or the closest to having sufficient getters, unless it is bettered by a bucket with potential alternate nodes.

If there are no good buckets we return the least_bad bucket.

Returns An instance of *ECGetResponseBucket* or None if there are no buckets in the collection.

choose_best_bucket()

property durable

get_extra_headers()

has_alternate_node()

property least_bad_bucket

Return the bad_bucket with the smallest shortfall

provide_alternate_node()

Callback function that is installed in a NodeIter. Called on every call to NodeIter.next(), which means we can track the number of nodes to which GET requests have been made and selectively inject an alternate node, if we have one.

Returns A dict describing a node to which the next GET request should be made.

property shortfall

class swift.proxy.controllers.obj.**ECObjectController**(*app*, *account_name*, *container_name*, *object_name*, ***kwargs*)

Bases: *swift.proxy.controllers.obj.BaseObjectController*

feed_remaining_primaries(*safe_iter*, *pile*, *req*, *partition*, *policy*, *buckets*, *feeder_q*, *logger_thread_locals*)

policy_type = 'erasure_coding'

class swift.proxy.controllers.obj.**MIMEPutter**(*conn*, *node*, *resp*, *req*, *connect_duration*, *watchdog*, *write_timeout*, *send_exception_handler*, *logger*, *mime_boundary*, *multiphase=False*)

Bases: *swift.proxy.controllers.obj.Putter*

Putter for backend PUT requests that use MIME.

This is here mostly to wrap up the fact that all multipart PUTs are chunked because of the mime boundary footer trick and the first half of the two-phase PUT conversation handling.

An HTTP PUT request that supports streaming.

classmethod connect(*node*, *part*, *req*, *headers*, *watchdog*, *conn_timeout*, *node_timeout*, *write_timeout*, *send_exception_handler*, *logger=None*, *need_multiphase=True*, ***kwargs*)

Connect to a backend node and send the headers.

Override superclass method to notify object of need for support for multipart body with footers and optionally multiphase commit, and verify object servers capabilities.

Parameters **need_multiphase** if True then multiphase support is required of the object server

Raises

- *FooterNotSupported* if need_metadata_footer is set but backend node cant process footers
- *MultiphasePUTNotSupported* if need_multiphase is set but backend node cant handle multiphase PUT

end_of_object_data(*footer_metadata=None*)

Call when there is no more data to send.

Overrides superclass implementation to send any footer metadata after object data.

Parameters **footer_metadata** dictionary of metadata items to be sent as footers.

send_commit_confirmation()

Call when there are > quorum 2XX responses received. Send commit confirmations to all object nodes to finalize the PUT.

class swift.proxy.controllers.obj.**ObjectControllerRouter**

Bases: object

policy_type_to_controller_map = {'erasure_coding': <class 'swift.proxy.controllers.obj.ECObjectController'>, 'replication': <class 'swift.proxy.controllers.obj.ReplicatedObjectController'>}

classmethod register(*policy_type*)

Decorator for Storage Policy implementations to register their ObjectController implementations.

This also fills in a policy_type attribute on the class.

class swift.proxy.controllers.obj.**Putter**(*conn*, *node*, *resp*, *path*, *connect_duration*, *watchdog*, *write_timeout*, *send_exception_handler*, *logger*, *chunked=False*)

Bases: object

Putter for backend PUT requests.

Encapsulates all the actions required to establish a connection with a storage node and stream data to that node.

Parameters

- **conn** an HTTPConnection instance
- **node** dict describing storage node
- **resp** an HTTPResponse instance if connect() received final response
- **path** the object path to send to the storage node
- **connect_duration** time taken to initiate the HTTPConnection
- **watchdog** a spawned Watchdog instance that will enforce timeouts
- **write_timeout** time limit to write a chunk to the connection socket
- **send_exception_handler** callback called when an exception occured writing to the connection socket
- **logger** a Logger instance
- **chunked** boolean indicating if the request encoding is chunked

await_response(*timeout*, *informational=False*)

Get 100-continue response indicating the end of 1st phase of a 2-phase commit or the final response, i.e. the one with status >= 200.

Might or might not actually wait for anything. If we said Expect: 100-continue but got back a non-100 response, thatll be the thing returned, and we wont do any network IO to get it. OTOH, if we got a 100 Continue response and sent up the PUT requests body, then well actually read the 2xx-5xx response off the network here.

Parameters

- **timeout** time to wait for a response
- **informational** if True then try to get a 100-continue response, otherwise try to get a final response.

Returns HTTPResponse

Raises **Timeout** if the response took too long

close()

classmethod **connect**(*node*, *part*, *path*, *headers*, *watchdog*, *conn_timeout*, *node_timeout*, *write_timeout*, *send_exception_handler*, *logger=None*, *chunked=False*, ***kwargs*)

Connect to a backend node and send the headers.

Returns Putter instance

Raises

- *ConnectionTimeout* if initial connection timed out
- *ResponseTimeout* if header retrieval timed out
- *InsufficientStorage* on 507 response from node
- *PutterConnectError* on non-507 server error response from node

end_of_object_data(***kwargs*)

Call when there is no more data to send.

send_chunk(*chunk*, *timeout_at=None*)

class swift.proxy.controllers.obj.**ReplicatedObjectController**(*app*, *account_name*, *container_name*, *object_name*, ***kwargs*)

Bases: *swift.proxy.controllers.obj.BaseObjectController*

policy_type = 'replication'

swift.proxy.controllers.obj.**check_content_type**(*req*)

swift.proxy.controllers.obj.**chunk_transformer**(*policy*)

A generator to transform a source chunk to erasure coded chunks for each *send* call. The number of erasure coded chunks is as policy.ec_n_unique_fragments.

swift.proxy.controllers.obj.**client_range_to_segment_range**(*client_start*, *client_end*, *segment_size*)

Takes a byterange from the client and converts it into a byterange spanning the necessary segments.

Handles prefix, suffix, and fully-specified byte ranges.

Examples: client_range_to_segment_range(100, 700, 512) = (0, 1023) client_range_to_segment_range(100, 700, 256) = (0, 767) client_range_to_segment_range(300, None, 256) = (256, None)

Parameters

- **client_start** first byte of the range requested by the client
- **client_end** last byte of the range requested by the client
- **segment_size** size of an EC segment, in bytes

Returns

a 2-tuple (seg_start, seg_end) where

- seg_start is the first byte of the first segment, or None if this is a suffix byte range
- seg_end is the last byte of the last segment, or None if this is a prefix byte range

swift.proxy.controllers.obj.**is_good_source**(*status*)

Indicates whether or not the request made to the backend found what it was looking for.

Parameters **status** the response from the backend

Returns True if found, False if not

swift.proxy.controllers.obj.**num_container_updates**(*container_replicas*, *container_quorum*, *object_replicas*, *object_quorum*)

We need to send container updates via enough object servers such that, if the object PUT succeeds, then the container update is durable (either its synchronously updated or written to async pendings).

Define: Qc = the quorum size for the container ring Qo = the quorum size for the object ring Rc = the replica count for the container ring Ro = the replica count (or EC N+K) for the object ring

A durable container update is one thats made it to at least Qc nodes. To always be durable, we have to send enough container updates so that, if only Qo object PUTs succeed, and all the failed object PUTs had container updates, at least Qc updates remain. Since (Ro - Qo) object PUTs may fail, we must have at least Qc + Ro - Qo container updates to ensure that Qc of them remain.

Also, each container replica is named in at least one object PUT request so that, when all requests succeed, no work is generated for the container replicator. Thus, at least Rc updates are necessary.

Parameters

- **container_replicas** replica count for the container ring (Rc)
- **container_quorum** quorum size for the container ring (Qc)
- **object_replicas** replica count for the object ring (Ro)
- **object_quorum** quorum size for the object ring (Qo)

swift.proxy.controllers.obj.**segment_range_to_fragment_range**(*segment_start*, *segment_end*, *segment_size*, *fragment_size*)

Takes a byterange spanning some segments and converts that into a byterange spanning the corresponding fragments within their fragment archives.

Handles prefix, suffix, and fully-specified byte ranges.

Parameters

- **segment_start** first byte of the first segment
- **segment_end** last byte of the last segment
- **segment_size** size of an EC segment, in bytes
- **fragment_size** size of an EC fragment, in bytes

Returns

a 2-tuple (frag_start, frag_end) where

- frag_start is the first byte of the first fragment, or None if this is a suffix byte range
- frag_end is the last byte of the last fragment, or None if this is a prefix byte range

swift.proxy.controllers.obj.**trailing_metadata**(*policy*, *client_obj_hasher*, *bytes_transferred_from_client*, *fragment_archive_index*)

9.2.2 Proxy Server

class swift.proxy.server.**Application**(*conf*, *logger=None*, *account_ring=None*, *container_ring=None*)

Bases: object

WSGI application for the proxy server.

check_config()

Check the configuration for possible errors

error_limit(*node*, *msg*)

Mark a node as error limited. This immediately pretends the node received enough errors to trigger error suppression. Use this for errors like Insufficient Storage. For other errors use *error_occurred()*.

Parameters

- **node** dictionary of node to error limit
- **msg** error message

error_limited(*node*)

Check if the node is currently error limited.

Parameters **node** dictionary of node to check

Returns True if error limited, False otherwise

error_occurred(*node*, *msg*)

Handle logging, and handling of errors.

Parameters

- **node** dictionary of node to handle errors for
- **msg** error message

exception_occurred(*node*, *typ*, *additional_info*, ***kwargs*)

Handle logging of generic exceptions.

Parameters

- **node** dictionary of node to log the error for
- **typ** server type
- **additional_info** additional information to log

get_controller(*req*)

Get the controller to handle a request.

Parameters **req** the request

Returns tuple of (controller class, path dictionary)

Raises `ValueError` (thrown by split_path) if given invalid path

`get_object_ring`(*policy_idx*)

Get the ring object to use to handle a request based on its policy.

Parameters `policy_idx` policy index as defined in swift.conf

Returns appropriate ring object

`get_policy_options`(*policy*)

Return policy specific options.

Parameters `policy` an instance of `BaseStoragePolicy` or `None`

Returns an instance of *ProxyOverrideOptions*

`handle_request`(*req*)

Entry point for proxy server. Should return a WSGI-style callable (such as swob.Response).

Parameters `req` swob.Request object

`iter_nodes`(*ring*, *partition*, *logger*, *node_iter=None*, *policy=None*)

`modify_wsgi_pipeline`(*pipe*)

Called during WSGI pipeline creation. Modifies the WSGI pipeline context to ensure that mandatory middleware is present in the pipeline.

Parameters `pipe` A PipelineWrapper object

`set_node_timing`(*node*, *timing*)

`sort_nodes`(*nodes*, *policy=None*)

Sorts nodes in-place (and returns the sorted list) according to the configured strategy. The default sorting is to randomly shuffle the nodes. If the timing strategy is chosen, the nodes are sorted according to the stored timing data.

Parameters

- `nodes` a list of nodes
- `policy` an instance of `BaseStoragePolicy`

`update_request`(*req*)

`class swift.proxy.server.ProxyOverrideOptions`(*base_conf*, *override_conf*, *app*)

Bases: `object`

Encapsulates proxy server options that may be overridden e.g. for policy specific configurations.

Parameters

- `conf` the proxy-server config dict.
- `override_conf` a dict of overriding configuration options.

`swift.proxy.server.app_factory`(*global_conf*, ***local_conf*)

paste.deploy app factory for creating WSGI proxy apps.

swift.proxy.server.**parse_per_policy_config**(*conf*)

Search the config file for any per-policy config sections and load those sections to a dict mapping policy reference (name or index) to policy options.

Parameters **conf** the proxy server conf dict

Returns a dict mapping policy reference -> dict of policy options

Raises **ValueError** if a policy config section has an invalid name

9.3 Account

9.3.1 Account Auditor

class swift.account.auditor.**AccountAuditor**(*conf*, *logger=None*)

Bases: swift.common.db_auditor.DatabaseAuditor

Audit accounts.

broker_class

alias of *swift.account.backend.AccountBroker*

server_type = 'account'

9.3.2 Account Backend

Pluggable Back-end for Account Server

class swift.account.backend.**AccountBroker**(*db_file*, *timeout=25*, *logger=None*, *account=None*, *container=None*, *pending_timeout=None*, *stale_reads_ok=False*, *skip_commits=False*)

Bases: *swift.common.db.DatabaseBroker*

Encapsulates working with an account database.

create_account_stat_table(*conn*, *put_timestamp*)

Create account_stat table which is specific to the account DB. Not a part of Pluggable Backends, internal to the baseline code.

Parameters

- **conn** DB connection object
- **put_timestamp** put timestamp

create_container_table(*conn*)

Create container table which is specific to the account DB.

Parameters **conn** DB connection object

create_policy_stat_table(*conn*)

Create policy_stat table which is specific to the account DB. Not a part of Pluggable Backends, internal to the baseline code.

Parameters **conn** DB connection object

db_contains_type = 'container'

db_reclaim_timestamp = 'delete_timestamp'

db_type = 'account'

empty()

Check if the account DB is empty.

Returns True if the database has no active containers.

get_db_version(*conn*)

get_info()

Get global data for the account.

Returns dict with keys: account, created_at, put_timestamp, delete_timestamp, status_changed_at, container_count, object_count, bytes_used, hash, id

get_policy_stats(*do_migrations=False*)

Get global policy stats for the account.

Parameters **do_migrations** boolean, if True the policy stat dicts will always include the container_count key; otherwise it may be omitted on legacy databases until they are migrated.

Returns dict of policy stats where the key is the policy index and the value is a dictionary like {object_count: M, bytes_used: N, container_count: L}

is_status_deleted()

Only returns true if the status field is set to DELETED.

list_containers_iter(*limit*, *marker*, *end_marker*, *prefix*, *delimiter*, *reverse=False*, *allow_reserved=False*)

Get a list of containers sorted by name starting at marker onward, up to limit entries. Entries will begin with the prefix and will not have the delimiter after the prefix.

Parameters

- **limit** maximum number of entries to get
- **marker** marker query
- **end_marker** end marker query
- **prefix** prefix query
- **delimiter** delimiter for query
- **reverse** reverse the result order.
- **allow_reserved** exclude names with reserved-byte by default

Returns list of tuples of (name, object_count, bytes_used, put_timestamp, 0)

make_tuple_for_pickle(*record*)

Turn this db record dict into the format this service uses for pending pickles.

merge_items(*item_list*, *source=None*)

Merge items into the container table.

Parameters

- **item_list** list of dictionaries of {name, put_timestamp, delete_timestamp, object_count, bytes_used, deleted, storage_policy_index}
- **source** if defined, update incoming_sync with the source

put_container(*name*, *put_timestamp*, *delete_timestamp*, *object_count*, *bytes_used*, *storage_policy_index*)

Create a container with the given attributes.

Parameters

- **name** name of the container to create (a native string)
- **put_timestamp** put_timestamp of the container to create
- **delete_timestamp** delete_timestamp of the container to create
- **object_count** number of objects in the container
- **bytes_used** number of bytes used by the container
- **storage_policy_index** the storage policy for this container

9.3.3 Account Reaper

class swift.account.reaper.**AccountReaper**(*conf*, *logger=None*)

Bases: swift.common.daemon.Daemon

Removes data from status=DELETED accounts. These are accounts that have been asked to be removed by the reseller via services remove_storage_account XMLRPC call.

The account is not deleted immediately by the services call, but instead the account is simply marked for deletion by setting the status column in the account_stat table of the account database. This account reaper scans for such accounts and removes the data in the background. The background deletion process will occur on the primary account server for the account.

Parameters

- **server_conf** The [account-server] dictionary of the account server configuration file
- **reaper_conf** The [account-reaper] dictionary of the account server configuration file

See the etc/account-server.conf-sample for information on the possible configuration parameters.

get_account_ring()

The account swift.common.ring.Ring for the cluster.

get_container_ring()

The container swift.common.ring.Ring for the cluster.

get_object_ring(*policy_idx*)

Get the ring identified by the policy index

Parameters **policy_idx** Storage policy index

Returns A ring matching the storage policy

reap_account(*broker*, *partition*, *nodes*, *container_shard=None*)

Called once per pass for each account this server is the primary for and attempts to delete the data for the given account. The reaper will only delete one account at any given time. It will call *reap_container()* up to sqrt(self.concurrency) times concurrently while reaping the account.

If there is any exception while deleting a single container, the process will continue for any other containers and the failed containers will be tried again the next time this function is called with the same parameters.

If there is any exception while listing the containers for deletion, the process will stop (but will obviously be tried again the next time this function is called with the same parameters). This isnt likely since the listing comes from the local database.

After the process completes (successfully or not) statistics about what was accomplished will be logged.

This function returns nothing and should raise no exception but only update various self.stats_* values for what occurs.

Parameters

- **broker** The AccountBroker for the account to delete.
- **partition** The partition in the account ring the account is on.
- **nodes** The primary node dicts for the account to delete.
- **container_shard** int used to shard containers reaped. If None, will reap all containers.

See also:

swift.account.backend.AccountBroker for the broker class.

See also:

`swift.common.ring.Ring.get_nodes()` for a description of the node dicts.

reap_container(*account*, *account_partition*, *account_nodes*, *container*)

Deletes the data and the container itself for the given container. This will call *reap_object()* up to sqrt(self.concurrency) times concurrently for the objects in the container.

If there is any exception while deleting a single object, the process will continue for any other objects in the container and the failed objects will be tried again the next time this function is called with the same parameters.

If there is any exception while listing the objects for deletion, the process will stop (but will obviously be tried again the next time this function is called with the same parameters). This is a possibility since the listing comes from querying just the primary remote container server.

Once all objects have been attempted to be deleted, the container itself will be attempted to be deleted by sending a delete request to all container nodes. The format of the delete request

is such that each container server will update a corresponding account server, removing the container from the accounts listing.

This function returns nothing and should raise no exception but only update various self.stats_* values for what occurs.

Parameters

- **account** The name of the account for the container.
- **account_partition** The partition for the account on the account ring.
- **account_nodes** The primary node dicts for the account.
- **container** The name of the container to delete.

- See also: `swift.common.ring.Ring.get_nodes()` for a description of the account node dicts.

reap_device(*device*)

Called once per pass for each device on the server. This will scan the accounts directory for the device, looking for partitions this device is the primary for, then looking for account databases that are marked status=DELETED and still have containers and calling *reap_account()*. Account databases marked status=DELETED that no longer have containers will eventually be permanently removed by the reclaim process within the account replicator (see `swift.db_replicator`).

Parameters **device** The device to look for accounts to be deleted.

reap_object(*account*, *container*, *container_partition*, *container_nodes*, *obj*, *policy_index*)

Deletes the given object by issuing a delete request to each node for the object. The format of the delete request is such that each object server will update a corresponding container server, removing the object from the containers listing.

This function returns nothing and should raise no exception but only update various self.stats_* values for what occurs.

Parameters

- **account** The name of the account for the object.
- **container** The name of the container for the object.
- **container_partition** The partition for the container on the container ring.
- **container_nodes** The primary node dicts for the container.
- **obj** The name of the object to delete.
- **policy_index** The storage policy index of the objects container

- See also: `swift.common.ring.Ring.get_nodes()` for a description of the container node dicts.

reset_stats()

run_forever(**args*, ***kwargs*)

Main entry point when running the reaper in normal daemon mode.

This repeatedly calls *run_once()* no quicker than the configuration interval.

run_once(**args*, ***kwargs*)

Main entry point when running the reaper in once mode, where it will do a single pass over all accounts on the server. This is called repeatedly by *run_forever()*. This will call *reap_device()* once for each device on the server.

9.3.4 Account Server

class swift.account.server.**AccountController**(*conf*, *logger=None*)

Bases: swift.common.base_storage_server.BaseStorageServer

WSGI controller for the account server.

DELETE(*req*)

Handle HTTP DELETE request.

GET(*req*)

Handle HTTP GET request.

HEAD(*req*)

Handle HTTP HEAD request.

POST(*req*)

Handle HTTP POST request.

PUT(*req*)

Handle HTTP PUT request.

REPLICATE(*req*)

Handle HTTP REPLICATE request. Handler for RPC calls for account replication.

check_free_space(*drive*)

server_type = 'account-server'

swift.account.server.**app_factory**(*global_conf*, ***local_conf*)

paste.deploy app factory for creating WSGI account server apps

swift.account.server.**get_account_name_and_placement**(*req*)

Split and validate path for an account.

Parameters **req** a swob request

Returns a tuple of path parts as strings

swift.account.server.**get_container_name_and_placement**(*req*)

Split and validate path for a container.

Parameters **req** a swob request

Returns a tuple of path parts as strings

9.4 Container

9.4.1 Container Auditor

class swift.container.auditor.**ContainerAuditor**(*conf*, *logger=None*)

Bases: swift.common.db_auditor.DatabaseAuditor

Audit containers.

broker_class

alias of *swift.container.backend.ContainerBroker*

server_type = 'container'

9.4.2 Container Backend

Pluggable Back-ends for Container Server

class swift.container.backend.**ContainerBroker**(*db_file*, *timeout=25*, *logger=None*, *account=None*, *container=None*, *pending_timeout=None*, *stale_reads_ok=False*, *skip_commits=False*, *force_db_file=False*)

Bases: *swift.common.db.DatabaseBroker*

Encapsulates working with a container database.

Note that this may involve multiple on-disk DB files if the container becomes sharded:

- _db_file is the path to the legacy container DB name, i.e. <hash>.db. This file should exist for an initialised broker that has never been sharded, but will not exist once a container has been sharded.
- *db_files* is a list of existing db files for the broker. This list should have at least one entry for an initialised broker, and should have two entries while a broker is in SHARDING state.
- *db_file* is the path to whichever db is currently authoritative for the container. Depending on the containers state, this may not be the same as the db_file argument given to __init__(), unless force_db_file is True in which case *db_file* is always equal to the db_file argument given to __init__().
- pending_file is always equal to _db_file extended with .pending, i.e. <hash>.db.pending.

classmethod create_broker(*device_path*, *part*, *account*, *container*, *logger=None*, *epoch=None*, *put_timestamp=None*, *storage_policy_index=None*)

Create a ContainerBroker instance. If the db doesnt exist, initialize the db file.

Parameters

- **device_path** device path
- **part** partition number

- **account** account name string
- **container** container name string
- **logger** a logger instance
- **epoch** a timestamp to include in the db filename
- **put_timestamp** initial timestamp if broker needs to be initialized
- **storage_policy_index** the storage policy index

Returns a tuple of (broker, initialized) where broker is an instance of *swift.container.backend.ContainerBroker* and initialized is True if the db file was initialized, False otherwise.

create_container_info_table(*conn*, *put_timestamp*, *storage_policy_index*)

Create the container_info table which is specific to the container DB. Not a part of Pluggable Back-ends, internal to the baseline code. Also creates the container_stat view.

Parameters

- **conn** DB connection object
- **put_timestamp** put timestamp
- **storage_policy_index** storage policy index

create_object_table(*conn*)

Create the object table which is specific to the container DB. Not a part of Pluggable Back-ends, internal to the baseline code.

Parameters **conn** DB connection object

create_policy_stat_table(*conn*, *storage_policy_index=0*)

Create policy_stat table.

Parameters

- **conn** DB connection object
- **storage_policy_index** the policy_index the container is being created with

create_shard_range_table(*conn*)

Create the shard_range table which is specific to the container DB.

Parameters **conn** DB connection object

db_contains_type = 'object'

property db_epoch

property db_file

Get the path to the primary db file for this broker. This is typically the db file for the most recent sharding epoch. However, if no db files exist on disk, or if force_db_file was True when the broker was constructed, then the primary db file is the file passed to the broker constructor.

Returns A path to a db file; the file does not necessarily exist.

property db_files

Gets the cached list of valid db files that exist on disk for this broker.

The cached list may be refreshed by calling *reload_db_files()*.

Returns A list of paths to db files ordered by ascending epoch; the list may be empty.

db_reclaim_timestamp = 'created_at'

db_type = 'container'

delete_meta_whitelist = ['x-container-sysmeta-shard-quoted-root', 'x-container-sysmeta-shard-root']

delete_object(*name*, *timestamp*, *storage_policy_index=0*)

Mark an object deleted.

Parameters

- **name** object name to be deleted
- **timestamp** timestamp when the object was marked as deleted
- **storage_policy_index** the storage policy index for the object

empty()

Check if container DB is empty.

This method uses more stringent checks on object count than `is_deleted()`: this method checks that there are no objects in any policy; if the container is in the process of sharding then both fresh and retiring databases are checked to be empty; if a root container has shard ranges then they are checked to be empty.

Returns True if the database has no active objects, False otherwise

enable_sharding(*epoch*)

Updates this brokers own shard range with the given epoch, sets its state to SHARDING and persists it in the DB.

Parameters **epoch** a `Timestamp`

Returns the brokers updated own shard range.

find_shard_ranges(*shard_size*, *limit=-1*, *existing_ranges=None*, *minimum_shard_size=1*)

Scans the container db for shard ranges. Scanning will start at the upper bound of the any `existing_ranges` that are given, otherwise at `ShardRange.MIN`. Scanning will stop when `limit` shard ranges have been found or when no more shard ranges can be found. In the latter case, the upper bound of the final shard range will be equal to the upper bound of the container namespace.

This method does not modify the state of the db; callers are responsible for persisting any shard range data in the db.

Parameters

- **shard_size** the size of each shard range

- **limit** the maximum number of shard points to be found; a negative value (default) implies no limit.
- **existing_ranges** an optional list of existing ShardRanges; if given, this list should be sorted in order of upper bounds; the scan for new shard ranges will start at the upper bound of the last existing ShardRange.
- **minimum_shard_size** Minimum size of the final shard range. If this is greater than one then the final shard range may be extended to more than shard_size in order to avoid a further shard range with less minimum_shard_size rows.

Returns a tuple; the first value in the tuple is a list of dicts each having keys {index, lower, upper, object_count} in order of ascending upper; the second value in the tuple is a boolean which is True if the last shard range has been found, False otherwise.

get_all_shard_range_data()

Returns a list of all shard range data, including own shard range and deleted shard ranges.

Returns A list of dict representations of a ShardRange.

get_brokers()

Return a list of brokers for component dbs. The list has two entries while the db state is sharding: the first entry is a broker for the retiring db with `skip_commits` set to `True`; the second entry is a broker for the fresh db with `skip_commits` set to `False`. For any other db state the list has one entry.

Returns a list of *ContainerBroker*

get_db_state()

Returns the current state of on disk db files.

get_db_version(*conn*)

get_info()

Get global data for the container.

Returns dict with keys: account, container, created_at, put_timestamp, delete_timestamp, status, status_changed_at, object_count, bytes_used, reported_put_timestamp, reported_delete_timestamp, reported_object_count, reported_bytes_used, hash, id, x_container_sync_point1, x_container_sync_point2, and storage_policy_index, db_state.

get_info_is_deleted()

Get the is_deleted status and info for the container.

Returns a tuple, in the form (info, is_deleted) info is a dict as returned by get_info and is_deleted is a boolean.

get_misplaced_since(*start*, *count*)

Get a list of objects which are in a storage policy different from the containers storage policy.

Parameters

- **start** last reconciler sync point
- **count** maximum number of entries to get

Returns list of dicts with keys: name, created_at, size, content_type, etag, storage_policy_index

get_objects(*limit=None*, *marker=''*, *end_marker=''*, *include_deleted=None*, *since_row=None*)

Returns a list of objects, including deleted objects, in all policies. Each object in the list is described by a dict with keys {name, created_at, size, content_type, etag, deleted, storage_policy_index}.

Parameters

- **limit** maximum number of entries to get
- **marker** if set, objects with names less than or equal to this value will not be included in the list.
- **end_marker** if set, objects with names greater than or equal to this value will not be included in the list.
- **include_deleted** if True, include only deleted objects; if False, include only undeleted objects; otherwise (default), include both deleted and undeleted objects.
- **since_row** include only items whose ROWID is greater than the given row id; by default all rows are included.

Returns a list of dicts, each describing an object.

get_own_shard_range(*no_default=False*)

Returns a shard range representing this brokers own shard range. If no such range has been persisted in the brokers shard ranges table then a default shard range representing the entire namespace will be returned.

The returned shard range will be updated with the current object stats for this broker and a meta timestamp set to the current time. For these values to be persisted the caller must merge the shard range.

Parameters **no_default** if True and the brokers own shard range is not found in the shard ranges table then None is returned, otherwise a default shard range is returned.

Returns an instance of *ShardRange*

get_policy_stats()

get_reconciler_sync()

get_replication_info()

Get information about the DB required for replication.

Returns dict containing keys from get_info plus max_row and metadata

Note:: get_infos <db_contains_type>_count is translated to just count and metadata is the raw string.

get_shard_ranges(*marker=None*, *end_marker=None*, *includes=None*, *reverse=False*, *include_deleted=False*, *states=None*, *include_own=False*, *exclude_others=False*, *fill_gaps=False*)

Returns a list of persisted shard ranges.

Parameters

- **marker** restricts the returned list to shard ranges whose namespace includes or is greater than the marker value.
- **end_marker** restricts the returned list to shard ranges whose namespace includes or is less than the end_marker value.
- **includes** restricts the returned list to the shard range that includes the given value; if `includes` is specified then `marker` and `end_marker` are ignored.
- **reverse** reverse the result order.
- **include_deleted** include items that have the delete marker set
- **states** if specified, restricts the returned list to shard ranges that have the given state(s); can be a list of ints or a single int.
- **include_own** boolean that governs whether the row whose name matches the brokers path is included in the returned list. If True, that row is included, otherwise it is not included. Default is False.
- **exclude_others** boolean that governs whether the rows whose names do not match the brokers path are included in the returned list. If True, those rows are not included, otherwise they are included. Default is False.
- **fill_gaps** if True, insert a modified copy of own shard range to fill any gap between the end of any found shard ranges and the upper bound of own shard range. Gaps enclosed within the found shard ranges are not filled.

Returns a list of instances of *swift.common.utils.ShardRange*

get_shard_usage()

Get the aggregate object stats for all shard ranges in states ACTIVE, SHARDING or SHRINKING.

Returns a dict with keys {bytes_used, object_count}

get_sharding_sysmeta(*key=None*)

Returns sharding specific info from the brokers metadata.

Parameters **key** if given the value stored under `key` in the sharding info will be returned.

Returns either a dict of sharding info or the value stored under `key` in that dict.

get_sharding_sysmeta_with_timestamps()

Returns sharding specific info from the brokers metadata with timestamps.

Parameters **key** if given the value stored under `key` in the sharding info will be returned.

Returns a dict of sharding info with their timestamps.

has_multiple_policies()

is_empty_enough_to_reclaim()

is_old_enough_to_reclaim(*now*, *reclaim_age*)

is_own_shard_range(*shard_range*)

is_reclaimable(*now*, *reclaim_age*)

Check if the broker abstraction is empty, and has been marked deleted for at least a reclaim age.

is_root_container()

Returns True if this container is a root container, False otherwise.

A root container is a container that is not a shard of another container.

is_sharded()

list_objects_iter(*limit*, *marker*, *end_marker*, *prefix*, *delimiter*, *path=None*, *storage_policy_index=0*, *reverse=False*, *include_deleted=False*, *since_row=None*, *transform_func=None*, *all_policies=False*, *allow_reserved=False*)

Get a list of objects sorted by name starting at marker onward, up to limit entries. Entries will begin with the prefix and will not have the delimiter after the prefix.

Parameters

- **limit** maximum number of entries to get
- **marker** marker query
- **end_marker** end marker query
- **prefix** prefix query
- **delimiter** delimiter for query
- **path** if defined, will set the prefix and delimiter based on the path
- **storage_policy_index** storage policy index for query
- **reverse** reverse the result order.
- **include_deleted** if True, include only deleted objects; if False (default), include only undeleted objects; otherwise, include both deleted and undeleted objects.
- **since_row** include only items whose ROWID is greater than the given row id; by default all rows are included.
- **transform_func** an optional function that if given will be called for each object to get a transformed version of the object to include in the listing; should have same signature as `_transform_record()`; defaults to `_transform_record()`.
- **all_policies** if True, include objects for all storage policies ignoring any value given for `storage_policy_index`
- **allow_reserved** exclude names with reserved-byte by default

Returns list of tuples of (name, created_at, size, content_type, etag, deleted)

make_tuple_for_pickle(*record*)

Turn this db record dict into the format this service uses for pending pickles.

merge_items(*item_list*, *source=None*)

Merge items into the object table.

Parameters

- **item_list** list of dictionaries of {name, created_at, size, content_type, etag, deleted, storage_policy_index, ctype_timestamp, meta_timestamp}
- **source** if defined, update incoming_sync with the source

merge_shard_ranges(*shard_ranges*)

Merge shard ranges into the shard range table.

Parameters **shard_ranges** a shard range or a list of shard ranges; each shard range should be an instance of *ShardRange* or a dict representation of a shard range having SHARD_RANGE_KEYS.

property path

put_object(*name*, *timestamp*, *size*, *content_type*, *etag*, *deleted=0*, *storage_policy_index=0*, *ctype_timestamp=None*, *meta_timestamp=None*)

Creates an object in the DB with its metadata.

Parameters

- **name** object name to be created
- **timestamp** timestamp of when the object was created
- **size** object size
- **content_type** object content-type
- **etag** object etag
- **deleted** if True, marks the object as deleted and sets the deleted_at timestamp to timestamp
- **storage_policy_index** the storage policy index for the object
- **ctype_timestamp** timestamp of when content_type was last updated
- **meta_timestamp** timestamp of when metadata was last updated

reload_db_files()

Reloads the cached list of valid on disk db files for this broker.

remove_objects(*lower*, *upper*, *max_row=None*)

Removes object records in the given namespace range from the object table.

Note that objects are removed regardless of their storage_policy_index.

Parameters

- **lower** defines the lower bound of object names that will be removed; names greater than this value will be removed; names less than or equal to this value will not be removed.

- **upper** defines the upper bound of object names that will be removed; names less than or equal to this value will be removed; names greater than this value will not be removed. The empty string is interpreted as there being no upper bound.
- **max_row** if specified only rows less than or equal to max_row will be removed

reported(*put_timestamp*, *delete_timestamp*, *object_count*, *bytes_used*)

Update reported stats, available with containers *get_info*.

Parameters

- **put_timestamp** put_timestamp to update
- **delete_timestamp** delete_timestamp to update
- **object_count** object_count to update
- **bytes_used** bytes_used to update

classmethod resolve_shard_range_states(*states*)

Given a list of values each of which may be the name of a state, the number of a state, or an alias, return the set of state numbers described by the list.

The following alias values are supported: listing maps to all states that are considered valid when listing objects; updating maps to all states that are considered valid for redirecting an object update; auditing maps to all states that are considered valid for a shard container that is updating its own shard range table from a root (this currently maps to all states except FOUND).

Parameters **states** a list of values each of which may be the name of a state, the number of a state, or an alias

Returns a set of integer state numbers, or None if no states are given

Raises **ValueError** if any value in the given list is neither a valid state nor a valid alias

property root_account

property root_container

property root_path

set_sharded_state()

Unlinks the brokers retiring DB file.

Returns True if the retiring DB was successfully unlinked, False otherwise.

set_sharding_state()

Creates and initializes a fresh DB file in preparation for sharding a retiring DB. The brokers own shard range must have an epoch timestamp for this method to succeed.

Returns True if the fresh DB was successfully created, False otherwise.

set_sharding_sysmeta(*key*, *value*)

Updates the brokers metadata stored under the given key prefixed with a sharding specific namespace.

Parameters

- **key** metadata key in the sharding metadata namespace.
- **value** metadata value

set_storage_policy_index(*policy_index*, *timestamp=None*)

Update the container_stat policy_index and status_changed_at.

set_x_container_sync_points(*sync_point1*, *sync_point2*)

sharding_initiated()

Returns True if a broker has shard range state that would be necessary for sharding to have been initiated, False otherwise.

sharding_required()

Returns True if a broker has shard range state that would be necessary for sharding to have been initiated but has not yet completed sharding, False otherwise.

property storage_policy_index

update_reconciler_sync(*point*)

swift.container.backend.**merge_shards**(*shard_data*, *existing*)

Compares shard_data with existing and updates shard_data with any items of existing that take precedence over the corresponding item in shard_data.

Parameters

- **shard_data** a dict representation of shard range that may be modified by this method.
- **existing** a dict representation of shard range.

Returns True if shard data has any item(s) that are considered to take precedence over the corresponding item in existing

swift.container.backend.**update_new_item_from_existing**(*new_item*, *existing*)

Compare the data and meta related timestamps of a new object item with the timestamps of an existing object record, and update the new item with data and/or meta related attributes from the existing record if their timestamps are newer.

The multiple timestamps are encoded into a single string for storing in the created_at column of the objects db table.

Parameters

- **new_item** A dict of object update attributes
- **existing** A dict of existing object attributes

Returns True if any attributes of the new item dict were found to be newer than the existing and therefore not updated, otherwise False implying that the updated item is equal to the existing.

9.4.3 Container Replicator

class swift.container.replicator.**ContainerReplicator**(*conf*, *logger=None*)

Bases: *swift.common.db_replicator.Replicator*

brokerclass

alias of *swift.container.backend.ContainerBroker*

cleanup_post_replicate(*broker*, *orig_info*, *responses*)

Cleanup non primary database from disk if needed.

Parameters

- **broker** the broker for the database were replicating
- **orig_info** snapshot of the broker replication info dict taken before replication
- **responses** a list of boolean success values for each replication request to other nodes

Return success returns False if deletion of the database was attempted but unsuccessful, otherwise returns True.

datadir = 'containers'

default_port = 6201

delete_db(*broker*)

Ensure that reconciler databases are only cleaned up at the end of the replication run.

dump_to_reconciler(*broker*, *point*)

Look for object rows for objects updates in the wrong storage policy in broker with a ROWID greater than the rowid given as point.

Parameters

- **broker** the container broker with misplaced objects
- **point** the last verified reconciler_sync_point

Returns the last successful enqueued rowid

feed_reconciler(*container*, *item_list*)

Add queue entries for rows in item_list to the local reconciler container database.

Parameters

- **container** the name of the reconciler container
- **item_list** the list of rows to enqueue

Returns True if successfully enqueued

find_local_handoff_for_part(*part*)

Find a device in the ring that is on this node on which to place a partition. Preference is given to a device that is a primary location for the partition. If no such device is found then a local device with weight is chosen, and failing that any local device.

Parameters **part** a partition

Returns a node entry from the ring

get_reconciler_broker(*timestamp*)

Get a local instance of the reconciler container broker that is appropriate to enqueue the given timestamp.

Parameters **timestamp** the timestamp of the row to be enqueued

Returns a local reconciler broker

replicate_reconcilers()

Ensure any items merged to reconciler containers during replication are pushed out to correct nodes and any reconciler containers that do not belong on this node are removed.

report_up_to_date(*full_info*)

run_once(**args*, ***kwargs*)

Run a replication pass once.

server_type = 'container'

class swift.container.replicator.**ContainerReplicatorRpc**(*root*, *datadir*, *broker_class*, *mount_check=True*, *logger=None*)

Bases: *swift.common.db_replicator.ReplicatorRpc*

get_shard_ranges(*broker*, *args*)

merge_shard_ranges(*broker*, *args*)

9.4.4 Container Server

class swift.container.server.**ContainerController**(*conf*, *logger=None*)

Bases: swift.common.base_storage_server.BaseStorageServer

WSGI Controller for the container server.

DELETE(*req*)

Handle HTTP DELETE request.

GET(*req*)

Handle HTTP GET request.

The body of the response to a successful GET request contains a listing of either objects or shard ranges. The exact content of the listing is determined by a combination of request headers and query string parameters, as follows:

- The type of the listing is determined by the X-Backend-Record-Type header. If this header has value shard then the response body will be a list of shard ranges; if this header has value auto, and the container state is sharding or sharded, then the listing will be a list of shard ranges; otherwise the response body will be a list of objects.
- Both shard range and object listings may be filtered according to the constraints described below. However, the X-Backend-Ignore-Shard-Name-Filter header may be used to override the application of the marker, end_marker, includes and reverse parameters to shard range listings. These parameters will be ignored if the

header has the value sharded and the current db sharding state is also sharded. Note that this header does not override the `states` constraint on shard range listings.

- The order of both shard range and object listings may be reversed by using a `reverse` query string parameter with a value in `swift.common.utils.TRUE_VALUES`.
- Both shard range and object listings may be constrained to a name range by the `marker` and `end_marker` query string parameters. Object listings will only contain objects whose names are greater than any `marker` value and less than any `end_marker` value. Shard range listings will only contain shard ranges whose namespace is greater than or includes any `marker` value and is less than or includes any `end_marker` value.
- Shard range listings may also be constrained by an `includes` query string parameter. If this parameter is present the listing will only contain shard ranges whose namespace includes the value of the parameter; any `marker` or `end_marker` parameters are ignored
- The length of an object listing may be constrained by the `limit` parameter. Object listings may also be constrained by `prefix`, `delimiter` and `path` query string parameters.
- Shard range listings will include deleted shard ranges if and only if the `X-Backend-Include-Deleted` header value is one of `swift.common.utils.TRUE_VALUES`. Object listings never include deleted objects.
- Shard range listings may be constrained to include only shard ranges whose state is specified by a query string `states` parameter. If present, the `states` parameter should be a comma separated list of either the string or integer representation of `STATES`.

  Two alias values may be used in a `states` parameter value: `listing` will cause the listing to include all shard ranges in a state suitable for contributing to an object listing; `updating` will cause the listing to include all shard ranges in a state suitable to accept an object update.

  If either of these aliases is used then the shard range listing will if necessary be extended with a synthesised filler range in order to satisfy the requested name range when insufficient actual shard ranges are found. Any filler shard range will cover the otherwise uncovered tail of the requested name range and will point back to the same container.
- Listings are not normally returned from a deleted container. However, the `X-Backend-Override-Deleted` header may be used with a value in `swift.common.utils.TRUE_VALUES` to force a shard range listing to be returned from a deleted container whose DB file still exists.

Parameters **req** an instance of *swift.common.swob.Request*

Returns an instance of *swift.common.swob.Response*

HEAD(*req*)

Handle HTTP HEAD request.

POST(*req*)

Handle HTTP POST request.

PUT(*req*)

Handle HTTP PUT request.

REPLICATE(*req*)

Handle HTTP REPLICATE request (json-encoded RPC calls for replication.)

UPDATE(*req*)

Handle HTTP UPDATE request (merge_items RPCs coming from the proxy.)

account_update(*req*, *account*, *container*, *broker*)

Update the account server(s) with latest container info.

Parameters

- **req** swob.Request object
- **account** account name
- **container** container name
- **broker** container DB broker object

Returns if all the account requests return a 404 error code, HTTPNotFound response object, if the account cannot be updated due to a malformed header, an HTTPBadRequest response object, otherwise None.

allowed_sync_hosts

The list of hosts were allowed to send syncs to. This can be overridden by data in self.realms_conf

check_free_space(*drive*)

create_listing(*req*, *out_content_type*, *info*, *resp_headers*, *metadata*, *container_list*, *container*)

get_and_validate_policy_index(*req*)

Validate that the index supplied maps to a policy.

Returns policy index from request, or None if not present

Raises **HTTPBadRequest** if the supplied index is bogus

realms_conf

ContainerSyncCluster instance for validating sync-to values.

save_headers = ['x-container-read', 'x-container-write', 'x-container-sync-key', 'x-container-sync-to']

server_type = 'container-server'

update_data_record(*record*)

Perform any mutations to container listing records that are common to all serialization formats, and returns it as a dict.

Converts created time to iso timestamp. Replaces size with swift_bytes content type parameter.

Params record object entry record

Returns modified record

swift.container.server.**app_factory**(*global_conf*, ***local_conf*)

paste.deploy app factory for creating WSGI container server apps

swift.container.server.**gen_resp_headers**(*info*, *is_deleted=False*)

Convert container info dict to headers.

swift.container.server.**get_container_name_and_placement**(*req*)

Split and validate path for a container.

Parameters **req** a swob request

Returns a tuple of path parts as strings

swift.container.server.**get_obj_name_and_placement**(*req*)

Split and validate path for an object.

Parameters **req** a swob request

Returns a tuple of path parts as strings

9.4.5 Container Reconciler

class swift.container.reconciler.**ContainerReconciler**(*conf*, *logger=None*, *swift=None*)

Bases: swift.common.daemon.Daemon

Move objects that are in the wrong storage policy.

can_reconcile_policy(*policy_index*)

ensure_object_in_right_location(*q_policy_index*, *account*, *container*, *obj*, *q_ts*, *path*, *container_policy_index*, *source_ts*, *source_obj_status*, *source_obj_info*, *source_obj_iter*, ***kwargs*)

Validate source object will satisfy the misplaced object queue entry and move to destination.

Parameters

- **q_policy_index** the policy_index for the source object
- **account** the account name of the misplaced object
- **container** the container name of the misplaced object
- **obj** the name of the misplaced object
- **q_ts** the timestamp of the misplaced object
- **path** the full path of the misplaced object for logging
- **container_policy_index** the policy_index of the destination
- **source_ts** the timestamp of the source object
- **source_obj_status** the HTTP status source object request
- **source_obj_info** the HTTP headers of the source object request
- **source_obj_iter** the body iter of the source object request

ensure_tombstone_in_right_location(*q_policy_index*, *account*, *container*, *obj*, *q_ts*, *path*, *container_policy_index*, *source_ts*, ***kwargs*)

Issue a DELETE request against the destination to match the misplaced DELETE against the source.

log_route = 'container-reconciler'

log_stats(*force=False*)

Dump stats to logger, noop when stats have been already been logged in the last minute.

pop_queue(*container*, *obj*, *q_ts*, *q_record*)

Issue a delete object request to the container for the misplaced object queue entry.

Parameters

- **container** the misplaced objects container
- **obj** the name of the misplaced object
- **q_ts** the timestamp of the misplaced object
- **q_record** the timestamp of the queue entry

N.B. q_ts will normally be the same time as q_record except when an object was manually re-enqued.

process_queue_item(*q_container*, *q_entry*, *queue_item*)

Process an entry and remove from queue on success.

Parameters

- **q_container** the queue container
- **q_entry** the raw_obj name from the q_container
- **queue_item** a parsed entry from the queue

reconcile()

Main entry point for concurrent processing of misplaced objects.

Iterate over all queue entries and delegate processing to spawned workers in the pool.

reconcile_object(*info*)

Process a possibly misplaced object write request. Determine correct destination storage policy by checking with primary containers. Check source and destination, copying or deleting into destination and cleaning up the source as needed.

This method wraps _reconcile_object for exception handling.

Parameters **info** a queue entry dict

Returns True to indicate the request is fully processed successfully, otherwise False.

run_forever(**args*, ***kwargs*)

Override this to run forever

run_once(**args*, ***kwargs*)

Process every entry in the queue.

should_process(*queue_item*)

Check if a given entry should be handled by this process.

Parameters

- **container** the queue container

- **queue_item** an entry from the queue

stats_log(*metric*, *msg*, **args*, ***kwargs*)

Update stats tracking for metric and emit log message.

throw_tombstones(*account*, *container*, *obj*, *timestamp*, *policy_index*, *path*)

Issue a delete object request to the given storage_policy.

Parameters

- **account** the account name
- **container** the container name
- **obj** the object name
- **timestamp** the timestamp of the object to delete
- **policy_index** the policy index to direct the request
- **path** the path to be used for logging

swift.container.reconciler.**add_to_reconciler_queue**(*container_ring*, *account*, *container*, *obj*, *obj_policy_index*, *obj_timestamp*, *op*, *force=False*, *conn_timeout=5*, *response_timeout=15*)

Add an object to the container reconcilers queue. This will cause the container reconciler to move it from its current storage policy index to the correct storage policy index.

Parameters

- **container_ring** container ring
- **account** the misplaced objects account
- **container** the misplaced objects container
- **obj** the misplaced object
- **obj_policy_index** the policy index where the misplaced object currently is
- **obj_timestamp** the misplaced objects X-Timestamp. We need this to ensure that the reconciler doesnt overwrite a newer object with an older one.
- **op** the method of the operation (DELETE or PUT)
- **force** over-write queue entries newer than obj_timestamp
- **conn_timeout** max time to wait for connection to container server
- **response_timeout** max time to wait for response from container server

Returns .misplaced_object container name, False on failure. Success means a majority of containers got the update.

swift.container.reconciler.**best_policy_index**(*headers*)

swift.container.reconciler.**cmp_policy_info**(*info*, *remote_info*)

You have to squint to see it, but the general strategy is just:

if either has been recreated: return the newest (of the recreated)

else return the oldest

I tried cleaning it up for awhile, but settled on just writing a bunch of tests instead. Once you get an intuitive sense for the nuance here you can try and see theres a better way to spell the boolean logic but it all ends up looking sorta hairy.

Returns -1 if info is correct, 1 if remote_info is better

swift.container.reconciler.**direct_delete_container_entry**(*container_ring*, *account_name*, *container_name*, *object_name*, *headers=None*)

Talk directly to the primary container servers to delete a particular object listing. Does not talk to object servers; use this only when a container entry does not actually have a corresponding object.

swift.container.reconciler.**get_reconciler_container_name**(*obj_timestamp*)

Get the name of a container into which a misplaced object should be enqueued. The name is the objects last modified time rounded down to the nearest hour.

Parameters **obj_timestamp** a string representation of the objects created_at time from its container db row.

Returns a container name

swift.container.reconciler.**get_reconciler_content_type**(*op*)

swift.container.reconciler.**get_reconciler_obj_name**(*policy_index*, *account*, *container*, *obj*)

swift.container.reconciler.**get_row_to_q_entry_translator**(*broker*)

swift.container.reconciler.**incorrect_policy_index**(*info*, *remote_info*)

Compare remote_info to info and decide if the remote storage policy index should be used instead of ours.

swift.container.reconciler.**parse_raw_obj**(*obj_info*)

Translate a reconciler container listing entry to a dictionary containing the parts of the misplaced object queue entry.

Parameters **obj_info** an entry in an a container listing with the required keys: name, content_type, and hash

Returns a queue entry dict with the keys: q_policy_index, account, container, obj, q_op, q_ts, q_record, and path

swift.container.reconciler.**slightly_later_timestamp**(*ts*, *offset=1*)

swift.container.reconciler.**translate_container_headers_to_info**(*headers*)

9.4.6 Container Sharder

class swift.container.sharder.**CleavingContext**(*ref*, *cursor=''*, *max_row=None*, *cleave_to_row=None*, *last_cleave_to_row=None*, *cleaving_done=False*, *misplaced_done=False*, *ranges_done=0*, *ranges_todo=0*)

Bases: object

Encapsulates metadata associated with the process of cleaving a retiring DB. This metadata includes:

- ref: The unique part of the key that is used when persisting a serialized CleavingContext as sysmeta in the DB. The unique part of the key is based off the DB id. This ensures that each context is associated with a specific DB file. The unique part of the key is included in the CleavingContext but should not be modified by any caller.
- cursor: the upper bound of the last shard range to have been cleaved from the retiring DB.
- max_row: the retiring DBs max row; this is updated to the value of the retiring DBs max_row every time a CleavingContext is loaded for that DB, and may change during the process of cleaving the DB.
- cleave_to_row: the value of max_row at the moment when cleaving starts for the DB. When cleaving completes (i.e. the cleave cursor has reached the upper bound of the cleaving namespace), cleave_to_row is compared to the current max_row: if the two values are not equal then rows have been added to the DB which may not have been cleaved, in which case the CleavingContext is reset and cleaving is re-started.
- last_cleave_to_row: the minimum DB row from which cleaving should select objects to cleave; this is initially set to None i.e. all rows should be cleaved. If the CleavingContext is reset then the last_cleave_to_row is set to the current value of cleave_to_row, which in turn is set to the current value of max_row by a subsequent call to start. The repeated cleaving therefore only selects objects in rows greater than the last_cleave_to_row, rather than cleaving the whole DB again.
- ranges_done: the number of shard ranges that have been cleaved from the retiring DB.
- ranges_todo: the number of shard ranges that are yet to be cleaved from the retiring DB.

property cursor

delete(*broker*)

done()

classmethod load(*broker*)

Returns a CleavingContext tracking the cleaving progress of the given brokers DB.

Parameters **broker** an instances of ContainerBroker

Returns An instance of *CleavingContext*.

classmethod load_all(*broker*)

Returns all cleaving contexts stored in the brokers DB.

Parameters **broker** an instance of ContainerBroker

Returns list of tuples of (CleavingContext, timestamp)

property marker

range_done(*new_cursor*)

reset()

start()

store(*broker*)

Persists the serialized `CleavingContext` as sysmeta in the given brokers DB.

Parameters **broker** an instances of `ContainerBroker`

class swift.container.sharder.**ContainerSharder**(*conf*, *logger=None*)

Bases: *swift.container.sharder.ContainerSharderConf*, *swift.container.replicator.ContainerReplicator*

Shards containers.

log_route = 'container-sharder'

run_forever(**args*, ***kwargs*)

Run the container sharder until stopped.

run_once(**args*, ***kwargs*)

Run the container sharder once.

yield_objects(*broker*, *src_shard_range*, *since_row=None*)

Iterates through all objects in `src_shard_range` in name order yielding them in lists of up to CONTAINER_LISTING_LIMIT length. Both deleted and undeleted objects are included.

Parameters

- **broker** A *ContainerBroker*.
- **src_shard_range** A *ShardRange* describing the source range.
- **since_row** include only items whose ROWID is greater than the given row id; by default all rows are included.

Returns a generator of tuples of (list of objects, broker info dict)

yield_objects_to_shard_range(*broker*, *src_shard_range*, *dest_shard_ranges*)

Iterates through all objects in `src_shard_range` to place them in destination shard ranges provided by the `next_shard_range` function. Yields tuples of (object list, destination shard range in which those objects belong). Note that the same destination shard range may be referenced in more than one yielded tuple.

Parameters

- **broker** A *ContainerBroker*.
- **src_shard_range** A *ShardRange* describing the source range.
- **dest_shard_ranges** A function which should return a list of destination shard ranges in name order.

Returns a generator of tuples of (object list, shard range, broker info dict)

class swift.container.sharder.**ContainerSharderConf**(*conf=None*)

Bases: object

percent_of_threshold(*val*)

classmethod **validate_conf**(*namespace*)

swift.container.sharder.**finalize_shrinking**(*broker*, *acceptor_ranges*, *donor_ranges*, *timestamp*)

Update donor shard ranges to shrinking state and merge donors and acceptors to broker.

Parameters

- **broker** A *ContainerBroker*.
- **acceptor_ranges** A list of *ShardRange* that are to be acceptors.
- **donor_ranges** A list of *ShardRange* that are to be donors; these will have their state and timestamp updated.
- **timestamp** timestamp to use when updating donor state

swift.container.sharder.**find_compactible_shard_sequences**(*broker*, *shrink_threshold*, *expansion_limit*, *max_shrinking*, *max_expanding*, *include_shrinking=False*)

Find sequences of shard ranges that could be compacted into a single acceptor shard range.

This function does not modify shard ranges.

Parameters

- **broker** A *ContainerBroker*.
- **shrink_threshold** the number of rows below which a shard may be considered for shrinking into another shard
- **expansion_limit** the maximum number of rows that an acceptor shard range should have after other shard ranges have been compacted into it
- **max_shrinking** the maximum number of shard ranges that should be compacted into each acceptor; -1 implies unlimited.
- **max_expanding** the maximum number of acceptors to be found (i.e. the maximum number of sequences to be returned); -1 implies unlimited.
- **include_shrinking** if True then existing compactible sequences are included in the results; default is False.

Returns A list of *ShardRangeList* each containing a sequence of neighbouring shard ranges that may be compacted; the final shard range in the list is the acceptor

swift.container.sharder.**find_overlapping_ranges**(*shard_ranges*)

Find all pairs of overlapping ranges in the given list.

Parameters **shard_ranges** A list of ShardRange

Returns a set of tuples, each tuple containing ranges that overlap with each other.

swift.container.sharder.**find_paths**(*shard_ranges*)

Returns a list of all continuous paths through the shard ranges. An individual path may not necessarily span the entire namespace, but it will span a continuous namespace without gaps.

Parameters **shard_ranges** A list of *ShardRange*.

Returns A list of *ShardRangeList*.

swift.container.sharder.**find_paths_with_gaps**(*shard_ranges*)

Find gaps in the shard ranges and pairs of shard range paths that lead to and from those gaps. For each gap a single pair of adjacent paths is selected. The concatenation of all selected paths and gaps will span the entire namespace with no overlaps.

Parameters **shard_ranges** a list of instances of ShardRange.

Returns A list of tuples of `(start_path, gap_range, end_path)` where `start_path` is a list of ShardRanges leading to the gap, `gap_range` is a ShardRange synthesized to describe the namespace gap, and `end_path` is a list of ShardRanges leading from the gap. When gaps start or end at the namespace minimum or maximum bounds, `start_path` and `end_path` may be null paths that contain a single ShardRange covering either the minimum or maximum of the namespace.

swift.container.sharder.**find_sharding_candidates**(*broker*, *threshold*, *shard_ranges=None*)

swift.container.sharder.**find_shrinking_candidates**(*broker*, *shrink_threshold*, *expansion_limit*)

swift.container.sharder.**is_sharding_candidate**(*shard_range*, *threshold*)

swift.container.sharder.**is_shrinking_candidate**(*shard_range*, *shrink_threshold*, *expansion_limit*, *states=None*)

swift.container.sharder.**make_shard_ranges**(*broker*, *shard_data*, *shards_account_prefix*)

swift.container.sharder.**process_compactible_shard_sequences**(*broker*, *sequences*)

Transform the given sequences of shard ranges into a list of acceptors and a list of shrinking donors. For each given sequence the final ShardRange in the sequence (the acceptor) is expanded to accommodate the other ShardRanges in the sequence (the donors). The donors and acceptors are then merged into the broker.

Parameters

- **broker** A *ContainerBroker*.
- **sequences** A list of *ShardRangeList*

swift.container.sharder.**random**() → x in the interval [0, 1).

swift.container.sharder.**rank_paths**(*paths*, *shard_range_to_span*)

Sorts the given list of paths such that the most preferred path is the first item in the list.

Parameters

- **paths** A list of *ShardRangeList*.

- **shard_range_to_span** An instance of *ShardRange* that describes the namespace that would ideally be spanned by a path. Paths that include this namespace will be preferred over those that do not.

Returns A sorted list of *ShardRangeList*.

swift.container.sharder.**sharding_enabled**(*broker*)

9.4.7 Container Sync

class swift.container.sync.**ContainerSync**(*conf*, *container_ring=None*, *logger=None*)

Bases: swift.common.daemon.Daemon

Daemon to sync syncable containers.

This is done by scanning the local devices for container databases and checking for x-container-sync-to and x-container-sync-key metadata values. If they exist, newer rows since the last sync will trigger PUTs or DELETEs to the other container.

The actual syncing is slightly more complicated to make use of the three (or number-of-replicas) main nodes for a container without each trying to do the exact same work but also without missing work if one node happens to be down.

Two sync points are kept per container database. All rows between the two sync points trigger updates. Any rows newer than both sync points cause updates depending on the nodes position for the container (primary nodes do one third, etc. depending on the replica count of course). After a sync run, the first sync point is set to the newest ROWID known and the second sync point is set to newest ROWID for which all updates have been sent.

An example may help. Assume replica count is 3 and perfectly matching ROWIDs starting at 1.

First sync run, database has 6 rows:

- SyncPoint1 starts as -1.
- SyncPoint2 starts as -1.
- No rows between points, so no all updates rows.
- Six rows newer than SyncPoint1, so a third of the rows are sent by node 1, another third by node 2, remaining third by node 3.
- SyncPoint1 is set as 6 (the newest ROWID known).
- SyncPoint2 is left as -1 since no all updates rows were synced.

Next sync run, database has 12 rows:

- SyncPoint1 starts as 6.
- SyncPoint2 starts as -1.
- The rows between -1 and 6 all trigger updates (most of which should short-circuit on the remote end as having already been done).
- Six more rows newer than SyncPoint1, so a third of the rows are sent by node 1, another third by node 2, remaining third by node 3.
- SyncPoint1 is set as 12 (the newest ROWID known).
- SyncPoint2 is set as 6 (the newest all updates ROWID).

In this way, under normal circumstances each node sends its share of updates each run and just sends a batch of older updates to ensure nothing was missed.

Parameters

- **conf** The dict of configuration values from the [container-sync] section of the container-server.conf
- **container_ring** If None, the <swift_dir>/container.ring.gz will be loaded. This is overridden by unit tests.

allowed_sync_hosts

The list of hosts were allowed to send syncs to. This can be overridden by data in self.realms_conf

conf

The dict of configuration values from the [container-sync] section of the container-server.conf.

container_deletes

Number of successful DELETEs triggered.

container_failures

Number of containers that had a failure of some type.

container_puts

Number of successful PUTs triggered.

container_report(*start*, *end*, *sync_point1*, *sync_point2*, *info*, *max_row*)

container_ring

swift.common.ring.Ring for locating containers.

container_skips

Number of containers whose sync has been turned off, but are not yet cleared from the sync store.

container_stats

Per container stats. These are collected per container. puts - the number of puts that were done for the container deletes - the number of deletes that were fot the container bytes - the total number of bytes transferred per the container

container_sync(*path*)

Checks the given path for a container database, determines if syncing is turned on for that database and, if so, sends any updates to the other container.

Parameters **path** the path to a container db

container_sync_row(*row*, *sync_to*, *user_key*, *broker*, *info*, *realm*, *realm_key*)

Sends the update the row indicates to the sync_to container. Update can be either delete or put.

Parameters

- **row** The updated row in the local database triggering the sync update.
- **sync_to** The URL to the remote container.

- **user_key** The X-Container-Sync-Key to use when sending requests to the other container.
- **broker** The local container database broker.
- **info** The get_info result from the local container database broker.
- **realm** The realm from self.realms_conf, if there is one. If None, fallback to using the older allowed_sync_hosts way of syncing.
- **realm_key** The realm key from self.realms_conf, if there is one. If None, fallback to using the older allowed_sync_hosts way of syncing.

Returns True on success

container_syncs

Number of containers with sync turned on that were successfully synced.

container_time

Maximum amount of time to spend syncing a container before moving on to the next one. If a container sync hasnt finished in this time, itll just be resumed next scan.

devices

Path to the local device mount points.

interval

Minimum time between full scans. This is to keep the daemon from running wild on near empty systems.

log_route = 'container-sync'

logger

Logger to use for container-sync log lines.

mount_check

Indicates whether mount points should be verified as actual mount points (normally true, false for tests and SAIO).

realms_conf

ContainerSyncCluster instance for validating sync-to values.

report()

Writes a report of the stats to the logger and resets the stats for the next report.

reported

Time of last stats report.

run_forever(**args*, ***kwargs*)

Runs container sync scans until stopped.

run_once(**args*, ***kwargs*)

Runs a single container sync scan.

select_http_proxy()

sync_store

ContainerSyncStore instance for iterating over synced containers

swift.container.sync.**random**() → x in the interval [0, 1).

9.4.8 Container Updater

class swift.container.updater.**ContainerUpdater**(*conf*, *logger=None*)

Bases: swift.common.daemon.Daemon

Update container information in account listings.

container_report(*node*, *part*, *container*, *put_timestamp*, *delete_timestamp*, *count*, *bytes*, *storage_policy_index*)

Report container info to an account server.

Parameters

- **node** node dictionary from the account ring
- **part** partition the account is on
- **container** container name
- **put_timestamp** put timestamp
- **delete_timestamp** delete timestamp
- **count** object count in the container
- **bytes** bytes used in the container
- **storage_policy_index** the policy index for the container

container_sweep(*path*)

Walk the path looking for container DBs and process them.

Parameters **path** path to walk

get_account_ring()

Get the account ring. Load it if it hasnt been yet.

get_paths()

Get paths to all of the partitions on each drive to be processed.

Returns a list of paths

process_container(*dbfile*)

Process a container, and update the information in the account.

Parameters **dbfile** container DB to process

run_forever(**args*, ***kwargs*)

Run the updater continuously.

run_once(**args*, ***kwargs*)

Run the updater once.

swift.container.updater.**random**() → x in the interval [0, 1).

9.5 Account DB and Container DB

9.5.1 DB

Database code for Swift

swift.common.db.**BROKER_TIMEOUT = 25**

Timeout for trying to connect to a DB

swift.common.db.**DB_PREALLOCATION = False**

Whether calls will be made to preallocate disk space for database files.

exception swift.common.db.**DatabaseAlreadyExists**(*path*)

Bases: sqlite3.DatabaseError

More friendly error messages for DB Errors.

class swift.common.db.**DatabaseBroker**(*db_file*, *timeout=25*, *logger=None*, *account=None*, *container=None*, *pending_timeout=None*, *stale_reads_ok=False*, *skip_commits=False*)

Bases: object

Encapsulates working with a database.

property db_file

delete_db(*timestamp*)

Mark the DB as deleted

Parameters **timestamp** internalized delete timestamp

delete_meta_whitelist = []

empty()

Check if the broker abstraction contains any undeleted records.

get()

Use with the with statement; returns a database connection.

get_device_path()

get_info()

get_items_since(*start*, *count*)

Get a list of objects in the database between start and end.

Parameters

- **start** start ROWID
- **count** number to get

Returns list of objects between start and end

get_max_row(*table=None*)

get_raw_metadata()

get_replication_info()

Get information about the DB required for replication.

Returns dict containing keys from get_info plus max_row and metadata

Note:: get_infos <db_contains_type>_count is translated to just count and metadata is the raw string.

get_sync(*id*, *incoming=True*)

Gets the most recent sync point for a server from the sync table.

Parameters

- **id** remote ID to get the sync_point for
- **incoming** if True, get the last incoming sync, otherwise get the last outgoing sync

Returns the sync point, or -1 if the id doesnt exist.

get_syncs(*incoming=True*)

Get a serialized copy of the sync table.

Parameters **incoming** if True, get the last incoming sync, otherwise get the last outgoing sync

Returns list of {remote_id, sync_point}

initialize(*put_timestamp=None*, *storage_policy_index=None*)

Create the DB

The storage_policy_index is passed through to the subclasss `_initialize` method. It is ignored by `AccountBroker`.

Parameters

- **put_timestamp** internalized timestamp of initial PUT request
- **storage_policy_index** only required for containers

is_deleted()

Check if the DB is considered to be deleted.

Returns True if the DB is considered to be deleted, False otherwise

is_reclaimable(*now*, *reclaim_age*)

Check if the broker abstraction is empty, and has been marked deleted for at least a reclaim age.

lock()

Use with the with statement; locks a database.

make_tuple_for_pickle(*record*)

Turn this db record dict into the format this service uses for pending pickles.

maybe_get(*conn*)

merge_items(*item_list*, *source=None*)

Save :param:item_list to the database.

merge_syncs(*sync_points*, *incoming=True*)

Merge a list of sync points with the incoming sync table.

Parameters

- **sync_points** list of sync points where a sync point is a dict of {sync_point, remote_id}
- **incoming** if True, get the last incoming sync, otherwise get the last outgoing sync

merge_timestamps(*created_at*, *put_timestamp*, *delete_timestamp*)

Used in replication to handle updating timestamps.

Parameters

- **created_at** create timestamp
- **put_timestamp** put timestamp
- **delete_timestamp** delete timestamp

property metadata

Returns the metadata dict for the database. The metadata dict values are tuples of (value, timestamp) where the timestamp indicates when that key was set to that value.

newid(*remote_id*)

Re-id the database. This should be called after an rsync.

Parameters **remote_id** the ID of the remote database being rsynced in

possibly_quarantine(*exc_type*, *exc_value*, *exc_traceback*)

Checks the exception info to see if it indicates a quarantine situation (malformed or corrupted database). If not, the original exception will be reraised. If so, the database will be quarantined and a new sqlite3.DatabaseError will be raised indicating the action taken.

put_record(*record*)

Put a record into the DB. If the DB has an associated pending file with space then the record is appended to that file and a commit to the DB is deferred. If the DB is in-memory or its pending file is full then the record will be committed immediately.

Parameters **record** a record to be added to the DB.

Raises

- *DatabaseConnectionError* if the DB file does not exist or if `skip_commits` is True.
- *LockTimeout* if a timeout occurs while waiting to take a lock to write to the pending file.

quarantine(*reason*)

The database will be quarantined and a sqlite3.DatabaseError will be raised indicating the action taken.

reclaim(*age_timestamp*, *sync_timestamp*)

Delete reclaimable rows and metadata from the db.

By default this method will delete rows from the db_contains_type table that are marked deleted and whose created_at timestamp is < age_timestamp, and deletes rows from incoming_sync and outgoing_sync where the updated_at timestamp is < sync_timestamp. In addition, this calls the `_reclaim_metadata()` method.

Subclasses may reclaim other items by overriding `_reclaim()`.

Parameters

- **age_timestamp** max created_at timestamp of object rows to delete
- **sync_timestamp** max update_at timestamp of sync rows to delete

update_metadata(*metadata_updates*, *validate_metadata=False*)

Updates the metadata dict for the database. The metadata dict values are tuples of (value, timestamp) where the timestamp indicates when that key was set to that value. Key/values will only be overwritten if the timestamp is newer. To delete a key, set its value to (, timestamp). These empty keys will eventually be removed by *reclaim()*

update_put_timestamp(*timestamp*)

Update the put_timestamp. Only modifies it if it is greater than the current timestamp.

Parameters **timestamp** internalized put timestamp

update_status_changed_at(*timestamp*)

Update the status_changed_at field in the stat table. Only modifies status_changed_at if the timestamp is greater than the current status_changed_at timestamp.

Parameters **timestamp** internalized timestamp

updated_timeout(*new_timeout*)

Use with with statement; updates `timeout` within the block.

static validate_metadata(*metadata*)

Validates that metadata falls within acceptable limits.

Parameters **metadata** to be validated

Raises **HTTPBadRequest** if MAX_META_COUNT or MAX_META_OVERALL_SIZE is exceeded, or if metadata contains non-UTF-8 data

exception swift.common.db.**DatabaseConnectionError**(*path*, *msg*, *timeout=0*)

Bases: `sqlite3.DatabaseError`

More friendly error messages for DB Errors.

class swift.common.db.**GreenDBConnection**(*database*, *timeout=None*, **args*, ***kwargs*)

Bases: `sqlite3.Connection`

SQLite DB Connection handler that plays well with eventlet.

commit()

Commit the current transaction.

cursor(*cls=None*)

Return a cursor for the connection.

class swift.common.db.**GreenDBCursor**(**args*, ***kwargs*)

Bases: sqlite3.Cursor

SQLite Cursor handler that plays well with eventlet.

execute(**args*, ***kwargs*)

Executes a SQL statement.

swift.common.db.**PICKLE_PROTOCOL = 2**

Pickle protocol to use

swift.common.db.**QUERY_LOGGING = False**

Whether calls will be made to log queries (py3 only)

class swift.common.db.**TombstoneReclaimer**(*broker*, *age_timestamp*)

Bases: object

Encapsulates reclamation of deleted rows in a database.

get_tombstone_count()

Return the number of remaining tombstones newer than age_timestamp. Executes the reclaim method if it has not already been called on this instance.

Returns The number of tombstones in the broker that are newer than age_timestamp.

reclaim()

Perform reclaim of deleted rows older than age_timestamp.

swift.common.db.**chexor**(*old*, *name*, *timestamp*)

Each entry in the account and container databases is XORed by the 128-bit hash on insert or delete. This serves as a rolling, order-independent hash of the contents. (check + XOR)

Parameters

- **old** hex representation of the current DB hash
- **name** name of the object or container being inserted
- **timestamp** internalized timestamp of the new record

Returns a hex representation of the new hash value

swift.common.db.**dict_factory**(*crs*, *row*)

This should only be used when you need a real dict, i.e. when youre going to serialize the results.

swift.common.db.**get_db_connection**(*path*, *timeout=30*, *logger=None*, *okay_to_create=False*)

Returns a properly configured SQLite database connection.

Parameters

- **path** path to DB
- **timeout** timeout for connection
- **okay_to_create** if True, create the DB if it doesnt exist

Returns DB connection object

swift.common.db.**native_str_keys_and_values**(*metadata*)

swift.common.db.**utf8encode**(**args*)

swift.common.db.**zero_like**(*count*)

Weve cargo culted our consumers to be tolerant of various expressions of zero in our databases for backwards compatibility with less disciplined producers.

9.5.2 DB replicator

class swift.common.db_replicator.**ReplConnection**(*node*, *partition*, *hash_*, *logger*)

Bases: *swift.common.bufferedhttp.BufferedHTTPConnection*

Helper to simplify REPLICATEing to a remote server.

replicate(**args*)

Make an HTTP REPLICATE request

Parameters **args** list of json-encodable objects

Returns bufferedhttp response object

class swift.common.db_replicator.**Replicator**(*conf*, *logger=None*)

Bases: swift.common.daemon.Daemon

Implements the logic for directing db replication.

cleanup_post_replicate(*broker*, *orig_info*, *responses*)

Cleanup non primary database from disk if needed.

Parameters

- **broker** the broker for the database were replicating
- **orig_info** snapshot of the broker replication info dict taken before replication
- **responses** a list of boolean success values for each replication request to other nodes

Return success returns False if deletion of the database was attempted but unsuccessful, otherwise returns True.

delete_db(*broker*)

extract_device(*object_file*)

Extract the device name from an object path. Returns UNKNOWN if the path could not be extracted successfully for some reason.

Parameters **object_file** the path to a database file.

report_up_to_date(*full_info*)

roundrobin_datadirs(*dirs*)

run_forever(**args*, ***kwargs*)

Replicate dbs under the given root in an infinite loop.

run_once(**args*, ***kwargs*)

Run a replication pass once.

class swift.common.db_replicator.**ReplicatorRpc**(*root*, *datadir*, *broker_class*, *mount_check=True*, *logger=None*)

Bases: object

Handle Replication RPC calls. TODO(redbo): document please :)

complete_rsync(*drive*, *db_file*, *args*)

debug_timing(*name*)

dispatch(*replicate_args*, *args*)

merge_items(*broker*, *args*)

merge_syncs(*broker*, *args*)

rsync_then_merge(*drive*, *db_file*, *args*)

sync(*broker*, *args*)

swift.common.db_replicator.**looks_like_partition**(*dir_name*)

True if the directory name is a valid partition number, False otherwise.

swift.common.db_replicator.**quarantine_db**(*object_file*, *server_type*)

In the case that a corrupt file is found, move it to a quarantined area to allow replication to fix it.

Parameters

- **object_file** path to corrupt file
- **server_type** type of file that is corrupt (container or account)

swift.common.db_replicator.**roundrobin_datadirs**(*datadirs*)

Generator to walk the data dirs in a round robin manner, evenly hitting each device on the system, and yielding any .db files found (in their proper places). The partitions within each data dir are walked randomly, however.

Parameters **datadirs** a list of tuples of (path, context, partition_filter) to walk. The context may be any object; the context is not used by this function but is included with each yielded tuple.

Returns A generator of (partition, path_to_db_file, context)

9.6 Object

9.6.1 Object Auditor

class swift.obj.auditor.**AuditorWorker**(*conf*, *logger*, *rcache*, *devices*, *zero_byte_only_at_fps=0*, *watcher_defs=None*)

Bases: object

Walk through file system to audit objects

audit_all_objects(*mode='once'*, *device_dirs=None*)

create_recon_nested_dict(*top_level_key*, *device_list*, *item*)

failsafe_object_audit(*location*)

Entrypoint to object_audit, with a failsafe generic exception handler.

object_audit(*location*)

Audits the given object location.

Parameters **location** an audit location (from diskfile.object_audit_location_generator)

record_stats(*obj_size*)

Based on configs object_size_stats will keep track of how many objects fall into the specified ranges. For example with the following:

object_size_stats = 10, 100, 1024

and your system has 3 objects of sizes: 5, 20, and 10000 bytes the log will look like: {10: 1, 100: 1, 1024: 0, OVER: 1}

class swift.obj.auditor.**ObjectAuditor**(*conf*, *logger=None*, ***options*)

Bases: swift.common.daemon.Daemon

Audit objects.

audit_loop(*parent*, *zbo_fps*, *override_devices=None*, ***kwargs*)

Parallel audit loop

clear_recon_cache(*auditor_type*)

Clear recon cache entries

fork_child(*zero_byte_fps=False*, *sleep_between_zbf_scanner=False*, ***kwargs*)

Child execution

run_audit(***kwargs*)

Run the object audit

run_forever(**args*, ***kwargs*)

Run the object audit until stopped.

run_once(**args*, ***kwargs*)

Run the object audit once

class swift.obj.auditor.**WatcherWrapper**(*watcher_class*, *watcher_name*, *conf*, *logger*)

Bases: object

Run the user-supplied watcher.

Simple and gets the job done. Note that we arent doing anything to isolate ourselves from hangs or file descriptor leaks in the plugins.

end()

see_object(*meta*, *data_file_path*)

start(*audit_type*)

9.6.2 Object Backend

Disk File Interface for the Swift Object Server

The *DiskFile*, *DiskFileWriter* and *DiskFileReader* classes combined define the on-disk abstraction layer for supporting the object server REST API interfaces (excluding *REPLICATE*). Other implementations wishing to provide an alternative backend for the object server must implement the three classes. An example alternative implementation can be found in the *mem_server.py* and *mem_diskfile.py* modules along size this one.

The *DiskFileManager* is a reference implemenation specific class and is not part of the backend API.

The remaining methods in this module are considered implementation specific and are also not considered part of the backend API.

class swift.obj.diskfile.**AuditLocation**(*path*, *device*, *partition*, *policy*)

Bases: object

Represents an object location to be audited.

Other than being a bucket of data, the only useful thing this does is stringify to a filesystem path so the auditors logs look okay.

class swift.obj.diskfile.**BaseDiskFile**(*mgr*, *device_path*, *partition*, *account=None*, *container=None*, *obj=None*, *_datadir=None*, *policy=None*, *use_splice=False*, *pipe_size=None*, *open_expired=False*, *next_part_power=None*, ***kwargs*)

Bases: object

Manage object files.

This specific implementation manages object files on a disk formatted with a POSIX-compliant file system that supports extended attributes as metadata on a file or directory.

Note: The arguments to the constructor are considered implementation specific. The API does not define the constructor arguments.

The following path format is used for data file locations: <devices_path/<device_dir>/<datadir>/<partdir>/<suffixdir>/<hashdir>/ <datafile>.<ext>

Parameters

- **mgr** associated DiskFileManager instance
- **device_path** path to the target device or drive
- **partition** partition on the device in which the object lives
- **account** account name for the object
- **container** container name for the object
- **obj** object name for the object
- **_datadir** override the full datadir otherwise constructed here
- **policy** the StoragePolicy instance

- **use_splice** if true, use zero-copy splice() to send data
- **pipe_size** size of pipe buffer used in zero-copy operations
- **open_expired** if True, open() will not raise a DiskFileExpired if object is expired
- **next_part_power** the next partition power to be used

property account

property container

property content_length

property content_type

property content_type_timestamp

create(*size=None*)

Context manager to create a file. We create a temporary file first, and then return a DiskFileWriter object to encapsulate the state.

Note: An implementation is not required to perform on-disk preallocations even if the parameter is specified. But if it does and it fails, it must raise a *DiskFileNoSpace* exception.

Parameters **size** optional initial size of file to explicitly allocate on disk

Raises *DiskFileNoSpace* if a size is specified and allocation fails

property data_timestamp

delete(*timestamp*)

Delete the object.

This implementation creates a tombstone file using the given timestamp, and removes any older versions of the object file. Any file that has an older timestamp than timestamp will be deleted.

Note: An implementation is free to use or ignore the timestamp parameter.

Parameters **timestamp** timestamp to compare with each file

Raises *DiskFileError* this implementation will raise the same errors as the *create()* method.

property durable_timestamp

Provides the timestamp of the newest data file found in the object directory.

Returns A Timestamp instance, or None if no data file was found.

Raises *DiskFileNotOpen* if the open() method has not been previously called on this instance.

property fragments

classmethod from_hash_dir(*mgr*, *hash_dir_path*, *device_path*, *partition*, *policy*)

get_datafile_metadata()

Provide the datafile metadata for a previously opened object as a dictionary. This is metadata that was included when the object was first PUT, and does not include metadata set by any subsequent POST.

Returns objects datafile metadata dictionary

Raises *DiskFileNotOpen* if the `swift.obj.diskfile.DiskFile.open()` method was not previously invoked

get_metadata()

Provide the metadata for a previously opened object as a dictionary.

Returns objects metadata dictionary

Raises *DiskFileNotOpen* if the `swift.obj.diskfile.DiskFile.open()` method was not previously invoked

get_metafile_metadata()

Provide the metafile metadata for a previously opened object as a dictionary. This is metadata that was written by a POST and does not include any persistent metadata that was set by the original PUT.

Returns objects .meta file metadata dictionary, or None if there is no .meta file

Raises *DiskFileNotOpen* if the `swift.obj.diskfile.DiskFile.open()` method was not previously invoked

property manager

property obj

open(*modernize=False*, *current_time=None*)

Open the object.

This implementation opens the data file representing the object, reads the associated metadata in the extended attributes, additionally combining metadata from fast-POST *.meta* files.

Parameters

- **modernize** if set, update this diskfile to the latest format. Currently, this means adding metadata checksums if none are present.
- **current_time** Unix time used in checking expiration. If not present, the current time will be used.

Note: An implementation is allowed to raise any of the following exceptions, but is only required to raise *DiskFileNotExist* when the object representation does not exist.

Raises

- *DiskFileCollision* on name mis-match with metadata
- *DiskFileNotExist* if the object does not exist

- *DiskFileDeleted* if the object was previously deleted
- *DiskFileQuarantined* if while reading metadata of the file some data did pass cross checks

Returns itself for use as a context manager

read_metadata(*current_time=None*)

Return the metadata for an object without requiring the caller to open the object first.

Parameters **current_time** Unix time used in checking expiration. If not present, the current time will be used.

Returns metadata dictionary for an object

Raises *DiskFileError* this implementation will raise the same errors as the *open()* method.

reader(*keep_cache=False*, *_quarantine_hook=<function BaseDiskFile.<lambda>*)

Return a *swift.common.swob.Response* class compatible *app_iter* object as defined by *swift.obj.diskfile.DiskFileReader*.

For this implementation, the responsibility of closing the open file is passed to the *swift.obj.diskfile.DiskFileReader* object.

Parameters

- **keep_cache** callers preference for keeping data read in the OS buffer cache
- **_quarantine_hook** 1-arg callable called when obj quarantined; the arg is the reason for quarantine. Default is to ignore it. Not needed by the REST layer.

Returns a *swift.obj.diskfile.DiskFileReader* object

reader_cls = None

property timestamp

write_metadata(*metadata*)

Write a block of metadata to an object without requiring the caller to create the object first. Supports fast-POST behavior semantics.

Parameters **metadata** dictionary of metadata to be associated with the object

Raises *DiskFileError* this implementation will raise the same errors as the *create()* method.

writer(*size=None*)

writer_cls = None

class swift.obj.diskfile.**BaseDiskFileManager**(*conf*, *logger*)

Bases: object

Management class for devices, providing common place for shared parameters and methods not provided by the DiskFile class (which primarily services the object server REST API layer).

The *get_diskfile()* method is how this implementation creates a *DiskFile* object.

Note: This class is reference implementation specific and not part of the pluggable on-disk backend API.

Note: TODO(portante): Not sure what the right name to recommend here, as manager seemed generic enough, though suggestions are welcome.

Parameters

- **conf** caller provided configuration object
- **logger** caller provided logger

classmethod check_policy(*policy*)

cleanup_ondisk_files(*hsh_path*, ***kwargs*)

Clean up on-disk files that are obsolete and gather the set of valid on-disk files for an object.

Parameters

- **hsh_path** object hash path
- **frag_index** if set, search for a specific fragment index .data file, otherwise accept the first valid .data file

Returns a dict that may contain: valid on disk files keyed by their filename extension; a list of obsolete files stored under the key obsolete; a list of files remaining in the directory, reverse sorted, stored under the key files.

clear_auditor_status(*policy*, *auditor_type='ALL'*)

static consolidate_hashes(*partition_dir*)

Take whats in hashes.pkl and hashes.invalid, combine them, write the result back to hashes.pkl, and clear out hashes.invalid.

Parameters **partition_dir** absolute path to partition dir containing hashes.pkl and hashes.invalid

Returns a dict, the suffix hashes (if any), the key valid will be False if hashes.pkl is corrupt, cannot be read or does not exist

construct_dev_path(*device*)

Construct the path to a device without checking if it is mounted.

Parameters **device** name of target device

Returns full path to the device

diskfile_cls = None

get_dev_path(*device*, *mount_check=None*)

Return the path to a device, first checking to see if either it is a proper mount point, or at least a directory depending on the mount_check configuration option.

Parameters

- **device** name of target device
- **mount_check** whether or not to check mountedness of device. Defaults to bool(self.mount_check).

Returns full path to the device, None if the path to the device is not a proper mount point or directory.

get_diskfile(*device*, *partition*, *account*, *container*, *obj*, *policy*, ***kwargs*)

Returns a BaseDiskFile instance for an object based on the objects partition, path parts and policy.

Parameters

- **device** name of target device
- **partition** partition on device in which the object lives
- **account** account name for the object
- **container** container name for the object
- **obj** object name for the object
- **policy** the StoragePolicy instance

get_diskfile_and_filenames_from_hash(*device*, *partition*, *object_hash*, *policy*, ***kwargs*)

Returns a tuple of (a DiskFile instance for an object at the given object_hash, the basenames of the files in the objects hash dir). Just in case someone thinks of refactoring, be sure DiskFileDeleted is *not* raised, but the DiskFile instance representing the tombstoned object is returned instead.

Parameters

- **device** name of target device
- **partition** partition on the device in which the object lives
- **object_hash** the hash of an object path
- **policy** the StoragePolicy instance

Raises *DiskFileNotExist* if the object does not exist

Returns a tuple comprising (an instance of BaseDiskFile, a list of file basenames)

get_diskfile_from_audit_location(*audit_location*)

Returns a BaseDiskFile instance for an object at the given AuditLocation.

Parameters **audit_location** object location to be audited

get_diskfile_from_hash(*device*, *partition*, *object_hash*, *policy*, ***kwargs*)

Returns a DiskFile instance for an object at the given object_hash. Just in case someone thinks of refactoring, be sure DiskFileDeleted is *not* raised, but the DiskFile instance representing the tombstoned object is returned instead.

Parameters

- **device** name of target device
- **partition** partition on the device in which the object lives

- **object_hash** the hash of an object path
- **policy** the StoragePolicy instance

Raises *DiskFileNotExist* if the object does not exist

Returns an instance of BaseDiskFile

get_hashes(*device*, *partition*, *suffixes*, *policy*, *skip_rehash=False*)

Parameters

- **device** name of target device
- **partition** partition name
- **suffixes** a list of suffix directories to be recalculated
- **policy** the StoragePolicy instance
- **skip_rehash** just mark the suffixes dirty; return None

Returns a dictionary that maps suffix directories

get_ondisk_files(*files*, *datadir*, *verify=True*, *policy=None*, ***kwargs*)

Given a simple list of files names, determine the files that constitute a valid fileset i.e. a set of files that defines the state of an object, and determine the files that are obsolete and could be deleted. Note that some files may fall into neither category.

If a file is considered part of a valid fileset then its info dict will be added to the results dict, keyed by <extension>_info. Any files that are no longer required will have their info dicts added to a list stored under the key obsolete.

The results dict will always contain entries with keys ts_file, data_file and meta_file. Their values will be the fully qualified path to a file of the corresponding type if there is such a file in the valid fileset, or None.

Parameters

- **files** a list of file names.
- **datadir** directory name files are from; this is used to construct file paths in the results, but the datadir is not modified by this method.
- **verify** if True verify that the ondisk file contract has not been violated, otherwise do not verify.
- **policy** storage policy used to store the files. Used to validate fragment indexes for EC policies.

Returns

a dict that will contain keys: ts_file -> path to a .ts file or None data_file -> path to a .data file or None meta_file -> path to a .meta file or None ctype_file -> path to a .meta file or None

and may contain keys: ts_info -> a file info dict for a .ts file data_info -> a file info dict for a .data file meta_info -> a file info dict for a .meta file ctype_info -> a file info dict for a .meta file which contains the content-type value unexpected -> a list of file paths for unexpected files possible_reclaim -> a list of file info dicts for possible reclaimable files obsolete -> a list of file info dicts for obsolete files

static invalidate_hash(*suffix_dir*)

Invalidates the hash for a suffix_dir in the partitions hashes file.

Parameters **suffix_dir** absolute path to suffix dir whose hash needs invalidating

make_on_disk_filename(*timestamp*, *ext=None*, *ctype_timestamp=None*, **a*, ***kw*)

Returns filename for given timestamp.

Parameters

- **timestamp** the object timestamp, an instance of *Timestamp*
- **ext** an optional string representing a file extension to be appended to the returned file name
- **ctype_timestamp** an optional content-type timestamp, an instance of *Timestamp*

Returns a file name

object_audit_location_generator(*policy*, *device_dirs=None*, *auditor_type='ALL'*)

Yield an AuditLocation for all objects stored under device_dirs.

Parameters

- **policy** the StoragePolicy instance
- **device_dirs** directory of target device
- **auditor_type** either ALL or ZBF

parse_on_disk_filename(*filename*, *policy*)

Parse an on disk file name.

Parameters

- **filename** the file name including extension
- **policy** storage policy used to store the file

Returns

a dict, with keys for timestamp, ext and ctype_timestamp:

- timestamp is a *Timestamp*
- ctype_timestamp is a *Timestamp* or None for .meta files, otherwise None
- ext is a string, the file extension including the leading dot or the empty string if the filename has no extension.

Subclasses may override this method to add further keys to the returned dict.

Raises *DiskFileError* if any part of the filename is not able to be validated.

partition_lock(*device*, *policy*, *partition*, *name=None*, *timeout=None*)

A context manager that will lock on the partition given.

Parameters

- **device** device targeted by the lock request
- **policy** policy targeted by the lock request

- **partition** partition targeted by the lock request

Raises *PartitionLockTimeout* If the lock on the partition cannot be granted within the configured timeout.

pickle_async_update(*device*, *account*, *container*, *obj*, *data*, *timestamp*, *policy*)

Write data describing a container update notification to a pickle file in the async_pending directory.

Parameters

- **device** name of target device
- **account** account name for the object
- **container** container name for the object
- **obj** object name for the object
- **data** update data to be written to pickle file
- **timestamp** a Timestamp
- **policy** the StoragePolicy instance

policy = None

static quarantine_renamer(*device_path*, *corrupted_file_path*)

In the case that a file is corrupted, move it to a quarantined area to allow replication to fix it.

Params device_path The path to the device the corrupted file is on.

Params corrupted_file_path The path to the file you want quarantined.

Returns path (str) of directory the file was moved to

Raises OSError re-raises non errno.EEXIST / errno.ENOTEMPTY exceptions from rename

replication_lock(*device*, *policy*, *partition*)

A context manager that will lock on the partition and, if configured to do so, on the device given.

Parameters

- **device** name of target device
- **policy** policy targeted by the replication request
- **partition** partition targeted by the replication request

Raises *ReplicationLockTimeout* If the lock on the device cannot be granted within the configured timeout.

yield_hashes(*device*, *partition*, *policy*, *suffixes=None*, ***kwargs*)

Yields tuples of (hash_only, timestamps) for object information stored for the given device, partition, and (optionally) suffixes. If suffixes is None, all stored suffixes will be searched for object hashes. Note that if suffixes is not None but empty, such as [], then nothing will be yielded.

timestamps is a dict which may contain items mapping:

- ts_data -> timestamp of data or tombstone file,

- ts_meta -> timestamp of meta file, if one exists
- **ts_ctype -> timestamp of meta file containing most recent** content-type value, if one exists
- durable -> True if data file at ts_data is durable, False otherwise

where timestamps are instances of *Timestamp*

Parameters

- **device** name of target device
- **partition** partition name
- **policy** the StoragePolicy instance
- **suffixes** optional list of suffix directories to be searched

yield_suffixes(*device*, *partition*, *policy*)

Yields tuples of (full_path, suffix_only) for suffixes stored on the given device and partition.

Parameters

- **device** name of target device
- **partition** partition name
- **policy** the StoragePolicy instance

class swift.obj.diskfile.**BaseDiskFileReader**(*fp*, *data_file*, *obj_size*, *etag*, *disk_chunk_size*, *keep_cache_size*, *device_path*, *logger*, *quarantine_hook*, *use_splice*, *pipe_size*, *diskfile*, *keep_cache=False*)

Bases: object

Encapsulation of the WSGI read context for servicing GET REST API requests. Serves as the context manager object for the *swift.obj.diskfile.DiskFile* classs `swift.obj.diskfile.DiskFile.reader()` method.

Note: The quarantining behavior of this method is considered implementation specific, and is not required of the API.

Note: The arguments to the constructor are considered implementation specific. The API does not define the constructor arguments.

Parameters

- **fp** open file object pointer reference
- **data_file** on-disk data file name for the object
- **obj_size** verified on-disk size of the object
- **etag** expected metadata etag value for entire file
- **disk_chunk_size** size of reads from disk in bytes

- **keep_cache_size** maximum object size that will be kept in cache
- **device_path** on-disk device path, used when quarantining an obj
- **logger** logger caller wants this object to use
- **quarantine_hook** 1-arg callable called w/reason when quarantined
- **use_splice** if true, use zero-copy splice() to send data
- **pipe_size** size of pipe buffer used in zero-copy operations
- **diskfile** the diskfile creating this DiskFileReader instance
- **keep_cache** should resulting reads be kept in the buffer cache

app_iter_range(*start*, *stop*)

Returns an iterator over the data file for range (start, stop)

app_iter_ranges(*ranges*, *content_type*, *boundary*, *size*)

Returns an iterator over the data file for a set of ranges

can_zero_copy_send()

close()

Close the open file handle if present.

For this specific implementation, this method will handle quarantining the file if necessary.

property manager

zero_copy_send(*wsockfd*)

Does some magic with splice() and tee() to move stuff from disk to network without ever touching userspace.

Parameters **wsockfd** file descriptor (integer) of the socket out which to send data

class swift.obj.diskfile.**BaseDiskFileWriter**(*name*, *datadir*, *size*, *bytes_per_sync*, *diskfile*, *next_part_power*)

Bases: object

Encapsulation of the write context for servicing PUT REST API requests. Serves as the context manager object for the *swift.obj.diskfile.DiskFile* classs swift.obj.diskfile.DiskFile.create() method.

Note: It is the responsibility of the swift.obj.diskfile.DiskFile.create() method context manager to close the open file descriptor.

Note: The arguments to the constructor are considered implementation specific. The API does not define the constructor arguments.

Parameters

- **name** name of object from REST API

- **datadir** on-disk directory object will end up in on *swift.obj.diskfile.DiskFileWriter.put()*
- **fd** open file descriptor of temporary file to receive data
- **tmppath** full path name of the opened file descriptor
- **bytes_per_sync** number bytes written between sync calls
- **diskfile** the diskfile creating this DiskFileWriter instance
- **next_part_power** the next partition power to be used

chunks_finished()

Expose internal stats about written chunks.

Returns a tuple, (upload_size, etag)

close()

commit(*timestamp*)

Perform any operations necessary to mark the object as durable. For replication policy type this is a no-op.

Parameters **timestamp** object put timestamp, an instance of *Timestamp*

property logger

property manager

open()

put(*metadata*)

Finalize writing the file on disk.

Parameters **metadata** dictionary of metadata to be associated with the object

write(*chunk*)

Write a chunk of data to disk. All invocations of this method must come before invoking the :func:

For this implementation, the data is written into a temporary file.

Parameters **chunk** the chunk of data to write as a string object

class swift.obj.diskfile.**DiskFile**(*mgr*, *device_path*, *partition*, *account=None*, *container=None*, *obj=None*, *_datadir=None*, *policy=None*, *use_splice=False*, *pipe_size=None*, *open_expired=False*, *next_part_power=None*, ***kwargs*)

Bases: *swift.obj.diskfile.BaseDiskFile*

reader_cls

alias of *swift.obj.diskfile.DiskFileReader*

writer_cls

alias of *swift.obj.diskfile.DiskFileWriter*

class swift.obj.diskfile.**DiskFileManager**(*conf*, *logger*)

Bases: *swift.obj.diskfile.BaseDiskFileManager*

diskfile_cls

alias of *swift.obj.diskfile.DiskFile*

policy = 'replication'

class swift.obj.diskfile.**DiskFileReader**(*fp*, *data_file*, *obj_size*, *etag*, *disk_chunk_size*, *keep_cache_size*, *device_path*, *logger*, *quarantine_hook*, *use_splice*, *pipe_size*, *diskfile*, *keep_cache=False*)

Bases: *swift.obj.diskfile.BaseDiskFileReader*

class swift.obj.diskfile.**DiskFileRouter**(**args*, ***kwargs*)

Bases: object

class swift.obj.diskfile.**DiskFileWriter**(*name*, *datadir*, *size*, *bytes_per_sync*, *diskfile*, *next_part_power*)

Bases: *swift.obj.diskfile.BaseDiskFileWriter*

put(*metadata*)

Finalize writing the file on disk.

Parameters **metadata** dictionary of metadata to be associated with the object

class swift.obj.diskfile.**ECDiskFile**(**args*, ***kwargs*)

Bases: *swift.obj.diskfile.BaseDiskFile*

property durable_timestamp

Provides the timestamp of the newest durable file found in the object directory.

Returns A Timestamp instance, or None if no durable file was found.

Raises *DiskFileNotOpen* if the open() method has not been previously called on this instance.

property fragments

Provides information about all fragments that were found in the object directory, including fragments without a matching durable file, and including any fragment chosen to construct the opened diskfile.

Returns A dict mapping <Timestamp instance> -> <list of frag indexes>, or None if the diskfile has not been opened or no fragments were found.

purge(*timestamp*, *frag_index*, *nondurable_purge_delay=0*, *meta_timestamp=None*)

Remove a tombstone file matching the specified timestamp or datafile matching the specified timestamp and fragment index from the object directory.

This provides the EC reconstructor/ssync process with a way to remove a tombstone or fragment from a handoff node after reverting it to its primary node.

The hash will be invalidated, and if empty the hsh_path will be removed immediately.

Parameters

- **timestamp** the object timestamp, an instance of *Timestamp*
- **frag_index** fragment archive index, must be a whole number or None.
- **nondurable_purge_delay** only remove a non-durable data file if its been on disk longer than this many seconds.

- **meta_timestamp** if not None then remove any meta file with this timestamp

reader_cls

alias of *swift.obj.diskfile.ECDiskFileReader*

writer_cls

alias of *swift.obj.diskfile.ECDiskFileWriter*

class swift.obj.diskfile.**ECDiskFileManager**(*conf*, *logger*)

Bases: *swift.obj.diskfile.BaseDiskFileManager*

diskfile_cls

alias of *swift.obj.diskfile.ECDiskFile*

make_on_disk_filename(*timestamp*, *ext=None*, *frag_index=None*, *ctype_timestamp=None*, *durable=False*, **a*, ***kw*)

Returns the EC specific filename for given timestamp.

Parameters

- **timestamp** the object timestamp, an instance of *Timestamp*
- **ext** an optional string representing a file extension to be appended to the returned file name
- **frag_index** a fragment archive index, used with .data extension only, must be a whole number.
- **ctype_timestamp** an optional content-type timestamp, an instance of *Timestamp*
- **durable** if True then include a durable marker in data filename.

Returns a file name

Raises *DiskFileError* if ext==.data and the kwarg frag_index is not a whole number

parse_on_disk_filename(*filename*, *policy*)

Returns timestamp(s) and other info extracted from a policy specific file name. For EC policy the data file name includes a fragment index and possibly a durable marker, both of which must be stripped off to retrieve the timestamp.

Parameters **filename** the file name including extension

Returns

a dict, with keys for timestamp, frag_index, durable, ext and ctype_timestamp:

- timestamp is a *Timestamp*
- frag_index is an int or None
- ctype_timestamp is a *Timestamp* or None for .meta files, otherwise None
- ext is a string, the file extension including the leading dot or the empty string if the filename has no extension
- durable is a boolean that is True if the filename is a data file that includes a durable marker

Raises *DiskFileError* if any part of the filename is not able to be validated.

policy = 'erasure_coding'

validate_fragment_index(*frag_index*, *policy=None*)

Return int representation of frag_index, or raise a DiskFileError if frag_index is not a whole number.

Parameters

- **frag_index** a fragment archive index
- **policy** storage policy used to validate the index against

class swift.obj.diskfile.**ECDiskFileReader**(*fp*, *data_file*, *obj_size*, *etag*, *disk_chunk_size*, *keep_cache_size*, *device_path*, *logger*, *quarantine_hook*, *use_splice*, *pipe_size*, *diskfile*, *keep_cache=False*)

Bases: *swift.obj.diskfile.BaseDiskFileReader*

class swift.obj.diskfile.**ECDiskFileWriter**(*name*, *datadir*, *size*, *bytes_per_sync*, *diskfile*, *next_part_power*)

Bases: *swift.obj.diskfile.BaseDiskFileWriter*

commit(*timestamp*)

Finalize put by renaming the object data file to include a durable marker. We do this for EC policy because it requires a 2-phase put commit confirmation.

Parameters **timestamp** object put timestamp, an instance of *Timestamp*

Raises *DiskFileError* if the diskfile frag_index has not been set (either during initialisation or a call to put())

put(*metadata*)

The only difference between this method and the replication policy DiskFileWriter method is adding the frag index to the metadata.

Parameters **metadata** dictionary of metadata to be associated with object

swift.obj.diskfile.**clear_auditor_status**(*devices*, *datadir*, *auditor_type='ALL'*)

swift.obj.diskfile.**consolidate_hashes**(*partition_dir*)

Take whats in hashes.pkl and hashes.invalid, combine them, write the result back to hashes.pkl, and clear out hashes.invalid.

Parameters **partition_dir** absolute path to partition dir containing hashes.pkl and hashes.invalid

Returns a dict, the suffix hashes (if any), the key valid will be False if hashes.pkl is corrupt, cannot be read or does not exist

swift.obj.diskfile.**extract_policy**(*obj_path*)

Extracts the policy for an object (based on the name of the objects directory) given the device-relative path to the object. Returns None in the event that the path is malformed in some way.

The device-relative path is everything after the mount point; for example:

/srv/node/d42/objects-5/30/179/ 485dc017205a81df3af616d917c90179/1401811134.873649.data

would have device-relative path:

objects-5/30/179/485dc017205a81df3af616d917c90179/1401811134.873649.data

Parameters **obj_path** device-relative path of an object, or the full path

Returns a *BaseStoragePolicy* or None

swift.obj.diskfile.**get_async_dir**(*policy_or_index*)

Get the async dir for the given policy.

Parameters **policy_or_index** StoragePolicy instance, or an index (string or int); if None, the legacy Policy-0 is assumed.

Returns async_pending or async_pending-<N> as appropriate

swift.obj.diskfile.**get_auditor_status**(*datadir_path*, *logger*, *auditor_type*)

swift.obj.diskfile.**get_data_dir**(*policy_or_index*)

Get the data dir for the given policy.

Parameters **policy_or_index** StoragePolicy instance, or an index (string or int); if None, the legacy Policy-0 is assumed.

Returns objects or objects-<N> as appropriate

swift.obj.diskfile.**get_part_path**(*dev_path*, *policy*, *partition*)

Given the device path, policy, and partition, returns the full path to the partition

swift.obj.diskfile.**get_tmp_dir**(*policy_or_index*)

Get the temp dir for the given policy.

Parameters **policy_or_index** StoragePolicy instance, or an index (string or int); if None, the legacy Policy-0 is assumed.

Returns tmp or tmp-<N> as appropriate

swift.obj.diskfile.**invalidate_hash**(*suffix_dir*)

Invalidates the hash for a suffix_dir in the partitions hashes file.

Parameters **suffix_dir** absolute path to suffix dir whose hash needs invalidating

swift.obj.diskfile.**object_audit_location_generator**(*devices*, *datadir*, *mount_check=True*, *logger=None*, *device_dirs=None*, *auditor_type='ALL'*)

Given a devices path (e.g. /srv/node), yield an AuditLocation for all objects stored under that directory for the given datadir (policy), if device_dirs isnt set. If device_dirs is set, only yield AuditLocation for the objects under the entries in device_dirs. The AuditLocation only knows the path to the hash directory, not to the .data file therein (if any). This is to avoid a double listdir(hash_dir); the DiskFile object will always do one, so we dont.

Parameters

- **devices** parent directory of the devices to be audited
- **datadir** objects directory
- **mount_check** flag to check if a mount check should be performed on devices
- **logger** a logger object

- **device_dirs** a list of directories under devices to traverse
- **auditor_type** either ALL or ZBF

swift.obj.diskfile.**quarantine_renamer**(*device_path*, *corrupted_file_path*)

In the case that a file is corrupted, move it to a quarantined area to allow replication to fix it.

Params device_path The path to the device the corrupted file is on.

Params corrupted_file_path The path to the file you want quarantined.

Returns path (str) of directory the file was moved to

Raises OSError re-raises non errno.EEXIST / errno.ENOTEMPTY exceptions from rename

swift.obj.diskfile.**read_hashes**(*partition_dir*)

Read the existing hashes.pkl

Returns a dict, the suffix hashes (if any), the key valid will be False if hashes.pkl is corrupt, cannot be read or does not exist

swift.obj.diskfile.**read_metadata**(*fd*, *add_missing_checksum=False*)

Helper function to read the pickled metadata from an object file.

Parameters

- **fd** file descriptor or filename to load the metadata from
- **add_missing_checksum** if set and checksum is missing, add it

Returns dictionary of metadata

swift.obj.diskfile.**relink_paths**(*target_path*, *new_target_path*, *ignore_missing=True*)

Hard-links a file located in `target_path` using the second path `new_target_path`. Creates intermediate directories if required.

Parameters

- **target_path** current absolute filename
- **new_target_path** new absolute filename for the hardlink
- **ignore_missing** if True then no exception is raised if the link could not be made because `target_path` did not exist, otherwise an OSError will be raised.

Raises OSError if the hard link could not be created, unless the intended hard link already exists or the `target_path` does not exist and `must_exist` if False.

Returns True if the link was created by the call to this method, False otherwise.

swift.obj.diskfile.**update_auditor_status**(*datadir_path*, *logger*, *partitions*, *auditor_type*)

swift.obj.diskfile.**valid_suffix**(*value*)

swift.obj.diskfile.**write_hashes**(*partition_dir*, *hashes*)

Write hashes to hashes.pkl

The updated key is added to hashes before it is written.

swift.obj.diskfile.**write_metadata**(*fd*, *metadata*, *xattr_size=65536*)

Helper function to write pickled metadata for an object file.

Parameters

- **fd** file descriptor or filename to write the metadata
- **metadata** metadata to write

9.6.3 Object Replicator

class swift.obj.replicator.**ObjectReplicator**(*conf*, *logger=None*)

Bases: swift.common.daemon.Daemon

Replicate objects.

Encapsulates most logic and data needed by the object replication process. Each call to .replicate() performs one replication pass. Its up to the caller to do this in a loop.

aggregate_recon_update()

build_replication_jobs(*policy*, *ips*, *override_devices=None*, *override_partitions=None*)

Helper function for collect_jobs to build jobs for replication using replication style storage policy

check_ring(*object_ring*)

Check to see if the ring has been updated :param object_ring: the ring to check

Returns boolean indicating whether or not the ring has changed

collect_jobs(*override_devices=None*, *override_partitions=None*, *override_policies=None*)

Returns a sorted list of jobs (dictionaries) that specify the partitions, nodes, etc to be rsynced.

Parameters

- **override_devices** if set, only jobs on these devices will be returned
- **override_partitions** if set, only jobs on these partitions will be returned
- **override_policies** if set, only jobs in these storage policies will be returned

delete_handoff_objs(*job*, *delete_objs*)

delete_partition(*path*)

get_local_devices()

Returns a set of all local devices in all replication-type storage policies.

This is the device names, e.g. sdq or d1234 or something, not the full ring entries.

get_worker_args(*once=False*, ***kwargs*)

For each worker yield a (possibly empty) dict of kwargs to pass along to the daemons run() method after fork. The length of elements returned from this method will determine the number of processes created.

If the returned iterable is empty, the Strategy will fallback to run-inline strategy.

Parameters

- **once** False if the worker(s) will be daemonized, True if the worker(s) will be run once
- **kwargs** plumbed through via command line argparser

Returns an iterable of dicts, each element represents the kwargs to be passed to a single workers `run()` method after fork.

heartbeat()

Loop that runs in the background during replication. It periodically logs progress.

is_healthy()

Check whether our set of local devices remains the same.

If devices have been added or removed, then we return False here so that we can kill off any worker processes and then distribute the new set of local devices across a new set of workers so that all devices are, once again, being worked on.

This function may also cause recon stats to be updated.

Returns False if any local devices have been added or removed, True otherwise

load_object_ring(*policy*)

Make sure the policys rings are loaded.

Parameters **policy** the StoragePolicy instance

Returns appropriate ring object

post_multiprocess_run()

Override this to do something after running using multiple worker processes. This method is called in the parent process.

This is probably only useful for run-once mode since there is no after running in run-forever mode.

replicate(*override_devices=None*, *override_partitions=None*, *override_policies=None*, *start_time=None*)

Run a replication pass

rsync(*node*, *job*, *suffixes*)

Uses rsync to implement the sync method. This was the first sync method in Swift.

run_forever(*multiprocess_worker_index=None*, *override_devices=None*, **args*, ***kwargs*)

Override this to run forever

run_once(*multiprocess_worker_index=None*, *have_overrides=False*, **args*, ***kwargs*)

Override this to run the script once

ssync(*node*, *job*, *suffixes*, *remote_check_objs=None*)

stats_line()

Logs various stats for the currently running replication pass.

sync(*node*, *job*, *suffixes*, **args*, ***kwargs*)

Synchronize local suffix directories from a partition with a remote node.

Parameters

- **node** the dev entry for the remote node to sync with
- **job** information about the partition being synced
- **suffixes** a list of suffixes which need to be pushed

Returns boolean and dictionary, boolean indicating success or failure

property total_stats

update(*job*)

High-level method that replicates a single partition.

Parameters **job** a dict containing info about the partition to be replicated

update_deleted(*job*)

High-level method that replicates a single partition that doesnt belong on this node.

Parameters **job** a dict containing info about the partition to be replicated

update_recon(*total*, *end_time*, *override_devices*)

class swift.obj.replicator.**Stats**(*attempted=0*, *failure=0*, *hashmatch=0*, *remove=0*, *rsync=0*, *success=0*, *suffix_count=0*, *suffix_hash=0*, *suffix_sync=0*, *failure_nodes=None*)

Bases: object

add_failure_stats(*failures*)

Note the failure of one or more devices.

Parameters **failures** a list of (ip, device-name) pairs that failed

fields = ['attempted', 'failure', 'hashmatch', 'remove', 'rsync', 'success', 'suffix_count', 'suffix_hash', 'suffix_sync', 'failure_nodes']

classmethod from_recon(*dct*)

to_recon()

class swift.obj.ssync_sender.**Sender**(*daemon*, *node*, *job*, *suffixes*, *remote_check_objs=None*, *include_non_durable=False*, *max_objects=0*)

Bases: object

Sends SSYNC requests to the object server.

These requests are eventually handled by *ssync_receiver* and full documentation about the process is there.

connect()

Establishes a connection and starts an SSYNC request with the object server.

disconnect(*connection*)

Closes down the connection to the object server once done with the SSYNC request.

missing_check(*connection*, *response*)

Handles the sender-side of the MISSING_CHECK step of a SSYNC request.

Full documentation of this can be found at *Receiver.missing_check()*.

send_delete(*connection*, *url_path*, *timestamp*)

Sends a DELETE subrequest with the given information.

send_post(*connection*, *url_path*, *df*)

send_put(*connection*, *url_path*, *df*, *durable=True*)

Sends a PUT subrequest for the url_path using the source df (DiskFile) and content_length.

send_subrequest(*connection*, *method*, *url_path*, *headers*, *df*)

updates(*connection*, *response*, *send_map*)

Handles the sender-side of the UPDATES step of an SSYNC request.

Full documentation of this can be found at *Receiver.updates()*.

class swift.obj.ssync_sender.**SsyncBufferedHTTPConnection**(*host*, *port=None*, *timeout=<object object>*, *source_address=None*)

Bases: *swift.common.bufferedhttp.BufferedHTTPConnection*

response_class

alias of *swift.obj.ssync_sender.SsyncBufferedHTTPResponse*

class swift.obj.ssync_sender.**SsyncBufferedHTTPResponse**(**args*, ***kwargs*)

Bases: *swift.common.bufferedhttp.BufferedHTTPResponse*, object

readline(*size=1024*)

Reads a line from the SSYNC response body.

httplib has no readline and will block on read(x) until x is read, so we have to do the work ourselves. A bit of this is taken from Pythons httplib itself.

swift.obj.ssync_sender.**decode_wanted**(*parts*)

Parse missing_check line parts to determine which parts of local diskfile were wanted by the receiver.

The encoder for parts is *encode_wanted()*

swift.obj.ssync_sender.**encode_missing**(*object_hash*, *ts_data*, *ts_meta=None*, *ts_ctype=None*, ***kwargs*)

Returns a string representing the object hash, its data file timestamp, the delta forwards to its metafile and content-type timestamps, if non-zero, and its durability, in the form: `<hash> <ts_data> [m:<hex delta to ts_meta>[,t:<hex delta to ts_ctype>] [,durable:False]`

The decoder for this line is *decode_missing()*

class swift.obj.ssync_receiver.**Receiver**(*app*, *request*)

Bases: object

Handles incoming SSYNC requests to the object server.

These requests come from the object-replicator daemon that uses *ssync_sender*.

The number of concurrent SSYNC requests is restricted by use of a replication_semaphore and can be configured with the object-server.conf [object-server] replication_concurrency setting.

An SSYNC request is really just an HTTP conduit for sender/receiver replication communication. The overall SSYNC request should always succeed, but it will contain multiple requests within its request and response bodies. This hack is done so that replication concurrency can be managed.

The general process inside an SSYNC request is:

1. Initialize the request: Basic request validation, mount check, acquire semaphore lock, etc..
2. Missing check: Sender sends the hashes and timestamps of the object information it can send, receiver sends back the hashes it wants (doesnt have or has an older timestamp).
3. Updates: Sender sends the object information requested.
4. Close down: Release semaphore lock, etc.

initialize_request()

Basic validation of request and mount check.

This function will be called before attempting to acquire a replication semaphore lock, so contains only quick checks.

missing_check()

Handles the receiver-side of the MISSING_CHECK step of a SSYNC request.

Receives a list of hashes and timestamps of object information the sender can provide and responds with a list of hashes desired, either because theyre missing or have an older timestamp locally.

The process is generally:

1. Sender sends `:MISSING_CHECK: START` and begins sending *hash timestamp* lines.
2. Receiver gets `:MISSING_CHECK: START` and begins reading the *hash timestamp* lines, collecting the hashes of those it desires.
3. Sender sends `:MISSING_CHECK: END`.
4. Receiver gets `:MISSING_CHECK: END`, responds with `:MISSING_CHECK: START`, followed by the list of <wanted_hash> specifiers it collected as being wanted (one per line), `:MISSING_CHECK: END`, and flushes any buffers.

   Each <wanted_hash> specifier has the form <hash>[<parts>] where <parts> is a string containing characters d and/or m indicating that only data or meta part of object respectively is required to be syncd.
5. Sender gets `:MISSING_CHECK: START` and reads the list of hashes desired by the receiver until reading `:MISSING_CHECK: END`.

The collection and then response is so the sender doesnt have to read while it writes to ensure network buffers dont fill up and block everything.

updates()

Handles the UPDATES step of an SSYNC request.

Receives a set of PUT and DELETE subrequests that will be routed to the object server itself for processing. These contain the information requested by the MISSING_CHECK step.

The PUT and DELETE subrequests are formatted pretty much exactly like regular HTTP requests, excepting the HTTP version on the first request line.

The process is generally:

1. Sender sends `:UPDATES: START` and begins sending the PUT and DELETE subrequests.
2. Receiver gets `:UPDATES: START` and begins routing the subrequests to the object server.
3. Sender sends `:UPDATES: END`.
4. Receiver gets `:UPDATES: END` and sends `:UPDATES: START` and `:UPDATES: END` (assuming no errors).
5. Sender gets `:UPDATES: START` and `:UPDATES: END`.

If too many subrequests fail, as configured by replication_failure_threshold and replication_failure_ratio, the receiver will hang up the request early so as to not waste any more time.

At step 4, the receiver will send back an error if there were any failures (that didnt cause a hangup due to the above thresholds) so the sender knows the whole was not entirely a success. This is so the sender knows if it can remove an out of place partition, for example.

exception swift.obj.ssync_receiver.**SsyncClientDisconnected**

Bases: Exception

swift.obj.ssync_receiver.**decode_missing**(*line*)

Parse a string of the form generated by *encode_missing()* and return a dict with keys object_hash, ts_data, ts_meta, ts_ctype, durable.

The encoder for this line is *encode_missing()*

swift.obj.ssync_receiver.**encode_wanted**(*remote*, *local*)

Compare a remote and local results and generate a wanted line.

Parameters

- **remote** a dict, with ts_data and ts_meta keys in the form returned by *decode_missing()*
- **local** a dict, possibly empty, with ts_data and ts_meta keys in the form returned Receiver._check_local()

The decoder for this line is *decode_wanted()*

9.6.4 Object Reconstructor

class swift.obj.reconstructor.**ObjectReconstructor**(*conf*, *logger=None*)

Bases: swift.common.daemon.Daemon

Reconstruct objects using erasure code. And also rebalance EC Fragment Archive objects off handoff nodes.

Encapsulates most logic and data needed by the object reconstruction process. Each call to .reconstruct() performs one pass. Its up to the caller to do this in a loop.

aggregate_recon_update()

Aggregate per-disk rcache updates from child workers.

build_reconstruction_jobs(*part_info*)

Helper function for collect_jobs to build jobs for reconstruction using EC style storage policy

N.B. If this function ever returns an empty list of jobs the entire partition will be deleted.

check_ring(*object_ring*)

Check to see if the ring has been updated

Parameters **object_ring** the ring to check

Returns boolean indicating whether or not the ring has changed

collect_parts(*override_devices=None*, *override_partitions=None*)

Helper for getting partitions in the top level reconstructor

In handoffs_only mode primary partitions will not be included in the returned (possibly empty) list.

delete_partition(*path*)

delete_reverted_objs(*job*, *objects*)

For EC we can potentially revert only some of a partition so well delete reverted objects here. Note that we delete the fragment index of the file we sent to the remote node.

Parameters

- **job** the job being processed
- **objects** a dict of objects to be deleted, each entry maps hash=>timestamp

detect_lockups()

In testing, the pool.waitall() call very occasionally failed to return. This is an attempt to make sure the reconstructor finishes its reconstruction pass in some eventuality.

final_recon_dump(*total*, *override_devices=None*, ***kwargs*)

Add stats for this workers run to recon cache.

When in worker mode (per_disk_stats == True) this workers stats are added per device instead of in the top level keys (aggregation is serialized in the parent process).

Parameters

- **total** the runtime of cycle in minutes
- **override_devices** (optional) list of device that are being reconstructed

get_local_devices()

Returns a set of all local devices in all EC policies.

get_policy2devices()

get_suffix_delta(*local_suff*, *local_index*, *remote_suff*, *remote_index*)

Compare the local suffix hashes with the remote suffix hashes for the given local and remote fragment indexes. Return those suffixes which should be synced.

Parameters

- **local_suff** the local suffix hashes (from _get_hashes)
- **local_index** the local fragment index for the job

- **remote_suff** the remote suffix hashes (from remote REPLICATE request)
- **remote_index** the remote fragment index for the job

Returns a list of strings, the suffix dirs to sync

get_worker_args(*once=False*, ***kwargs*)

Take the set of all local devices for this node from all the EC policies rings, and distribute them evenly into the number of workers to be spawned according to the configured worker count. If *devices* is given in *kwargs* then distribute only those devices.

Parameters

- **once** False if the worker(s) will be daemonized, True if the worker(s) will be run once
- **kwargs** optional overrides from the command line

heartbeat()

Loop that runs in the background during reconstruction. It periodically logs progress.

is_healthy()

Check whether rings have changed, and maybe do a recon update.

Returns False if any ec ring has changed

kill_coros()

Utility function that kills all coroutines currently running.

load_object_ring(*policy*)

Make sure the policys rings are loaded.

Parameters **policy** the StoragePolicy instance

Returns appropriate ring object

make_rebuilt_fragment_iter(*responses*, *path*, *policy*, *frag_index*)

Turn a set of connections from backend object servers into a generator that yields up the rebuilt fragment archive for frag_index.

post_multiprocess_run()

Override this to do something after running using multiple worker processes. This method is called in the parent process.

This is probably only useful for run-once mode since there is no after running in run-forever mode.

process_job(*job*)

Sync the local partition with the remote node(s) according to the parameters of the job. For primary nodes, the SYNC job type will define both left and right hand sync_to nodes to ssync with as defined by this primary nodes index in the node list based on the fragment index found in the partition. For non-primary nodes (either handoff revert, or rebalance) the REVERT job will define a single node in sync_to which is the proper/new home for the fragment index.

N.B. ring rebalancing can be time consuming and handoff nodes fragment indexes do not have a stable order, its possible to have more than one REVERT job for a partition, and in some rare failure conditions there may even also be a SYNC job for the same partition - but each one will be processed separately because each job will define a separate list of node(s) to sync_to.

Parameters **job** the job dict, with the keys defined in _get_job_info

reconstruct(***kwargs*)

Run a reconstruction pass

reconstruct_fa(*job*, *node*, *df*)

Reconstructs a fragment archive - this method is called from ssync after a remote node responds that is missing this object - the local diskfile is opened to provide metadata - but to reconstruct the missing fragment archive we must connect to multiple object servers.

Parameters

- **job** job from ssync_sender.
- **node** node to which were rebuilding.
- **df** an instance of *BaseDiskFile*.

Returns a DiskFile like class for use by ssync.

Raises

- *DiskFileQuarantined* if the fragment archive cannot be reconstructed and has as a result been quarantined.
- *DiskFileError* if the fragment archive cannot be reconstructed.

run_forever(*multiprocess_worker_index=None*, **args*, ***kwargs*)

Override this to run forever

run_once(*multiprocess_worker_index=None*, **args*, ***kwargs*)

Override this to run the script once

stats_line()

Logs various stats for the currently running reconstruction pass.

class swift.obj.reconstructor.**RebuildingECDiskFileStream**(*datafile_metadata*, *frag_index*, *rebuilt_fragment_iter*)

Bases: object

This class wraps the reconstructed fragment archive data and metadata in the DiskFile interface for ssync.

property content_length

get_datafile_metadata()

get_metadata()

reader()

class swift.obj.reconstructor.**ResponseBucket**

Bases: object

Encapsulates fragment GET response data related to a single timestamp.

9.6.5 Object Server

Object Server for Swift

class swift.obj.server.**EventletPlungerString**

Bases: bytes

Eventlet wont send headers until its accumulated at least eventlet.wsgi.MINIMUM_CHUNK_SIZE bytes or the app iter is exhausted. If we want to send the response body behind Eventlets back, perhaps with some zero-copy wizardry, then we have to unclog the plumbing in eventlet.wsgi to force the headers out, so we use an EventletPlungerString to empty out all of Eventlets buffers.

class swift.obj.server.**ObjectController**(*conf*, *logger=None*)

Bases: swift.common.base_storage_server.BaseStorageServer

Implements the WSGI application for the Swift Object Server.

DELETE(*request*)

Handle HTTP DELETE requests for the Swift Object Server.

GET(*request*)

Handle HTTP GET requests for the Swift Object Server.

HEAD(*request*)

Handle HTTP HEAD requests for the Swift Object Server.

POST(*request*)

Handle HTTP POST requests for the Swift Object Server.

PUT(*request*)

Handle HTTP PUT requests for the Swift Object Server.

REPLICATE(*request*)

Handle REPLICATE requests for the Swift Object Server. This is used by the object replicator to get hashes for directories.

Note that the name REPLICATE is preserved for historical reasons as this verb really just returns the hashes information for the specified parameters and is used, for example, by both replication and EC.

SSYNC(*request*)

async_update(*op*, *account*, *container*, *obj*, *host*, *partition*, *contdevice*, *headers_out*, *objdevice*, *policy*, *logger_thread_locals=None*, *container_path=None*)

Sends or saves an async update.

Parameters

- **op** operation performed (ex: PUT, or DELETE)
- **account** account name for the object
- **container** container name for the object
- **obj** object name
- **host** host that the container is on

- **partition** partition that the container is on
- **contdevice** device name that the container is on
- **headers_out** dictionary of headers to send in the container request
- **objdevice** device name that the object is in
- **policy** the associated BaseStoragePolicy instance
- **logger_thread_locals** The thread local values to be set on the self.logger to retain transaction logging information.
- **container_path** optional path in the form <*account/container*> to which the update should be sent. If given this path will be used instead of constructing a path from the `account` and `container` params.

container_update(*op*, *account*, *container*, *obj*, *request*, *headers_out*, *objdevice*, *policy*)

Update the container when objects are updated.

Parameters

- **op** operation performed (ex: PUT, or DELETE)
- **account** account name for the object
- **container** container name for the object
- **obj** object name
- **request** the original request object driving the update
- **headers_out** dictionary of headers to send in the container request(s)
- **objdevice** device name that the object is in
- **policy** the BaseStoragePolicy instance

delete_at_update(*op*, *delete_at*, *account*, *container*, *obj*, *request*, *objdevice*, *policy*)

Update the expiring objects container when objects are updated.

Parameters

- **op** operation performed (ex: PUT, or DELETE)
- **delete_at** scheduled delete in UNIX seconds, int
- **account** account name for the object
- **container** container name for the object
- **obj** object name
- **request** the original request driving the update
- **objdevice** device name that the object is in
- **policy** the BaseStoragePolicy instance (used for tmp dir)

get_diskfile(*device*, *partition*, *account*, *container*, *obj*, *policy*, ***kwargs*)

Utility method for instantiating a DiskFile object supporting a given REST API.

An implementation of the object server that wants to use a different DiskFile class would simply over-ride this method to provide that behavior.

server_type = 'object-server'

setup(*conf*)

Implementation specific setup. This method is called at the very end by the constructor to allow a specific implementation to modify existing attributes or add its own attributes.

Parameters **conf** WSGI configuration parameter

swift.obj.server.**app_factory**(*global_conf*, ***local_conf*)

paste.deploy app factory for creating WSGI object server apps

swift.obj.server.**drain**(*file_like*, *read_size*, *timeout*)

Read and discard any bytes from file_like.

Parameters

- **file_like** file-like object to read from
- **read_size** how big a chunk to read at a time
- **timeout** how long to wait for a read (use None for no timeout)

Raises *ChunkReadTimeout* if no chunk was read in time

swift.obj.server.**get_obj_name_and_placement**(*request*)

Split and validate path for an object.

Parameters **request** a swob request

Returns a tuple of path parts and storage policy

swift.obj.server.**global_conf_callback**(*preloaded_app_conf*, *global_conf*)

Callback for swift.common.wsgi.run_wsgi during the global_conf creation so that we can add our replication_semaphore, used to limit the number of concurrent SSYNC_REQUESTS across all workers.

Parameters

- **preloaded_app_conf** The preloaded conf for the WSGI app. This conf instance will go away, so just read from it, dont write.
- **global_conf** The global conf that will eventually be passed to the app_factory function later. This conf is created before the worker subprocesses are forked, so can be useful to set up semaphores, shared memory, etc.

swift.obj.server.**iter_mime_headers_and_bodies**(*wsgi_input*, *mime_boundary*, *read_chunk_size*)

9.6.6 Object Updater

class swift.obj.updater.**BucketizedUpdateSkippingLimiter**(*update_iterable*, *logger*, *stats*, *num_buckets=1000*, *max_elements_per_group_per_second=50*, *max_deferred_elements=0*, *drain_until=0*)

Bases: object

Wrap an iterator to rate-limit updates on a per-bucket basis, where updates are mapped to buckets by hashing their destination path. If an update is rate-limited then it is placed on a deferral queue and may be sent later if the wrapped iterator is exhausted before the `drain_until` time is reached.

The deferral queue has constrained size and once the queue is full updates are evicted using a first-in-first-out policy. This policy is used because updates on the queue may have been made obsolete by newer updates written to disk, and this is more likely for updates that have been on the queue longest.

The iterator increments stats as follows:

- The *deferrals* stat is incremented for each update that is rate-limited. Note that a individual update is rate-limited at most once.
- The *skips* stat is incremented for each rate-limited update that is not eventually yielded. This includes updates that are evicted from the deferral queue and all updates that remain in the deferral queue when `drain_until` time is reached and the iterator terminates.
- The *drains* stat is incremented for each rate-limited update that is eventually yielded.

Consequently, when this iterator terminates, the sum of *skips* and *drains* is equal to the number of *deferrals*.

Parameters

- **update_iterable** an async_pending update iterable
- **logger** a logger instance
- **stats** a SweepStats instance
- **num_buckets** number of buckets to divide container hashes into, the more buckets total the less containers to a bucket (once a busy container slows down a bucket the whole bucket starts deferring)
- **max_elements_per_group_per_second** tunable, when deferring kicks in
- **max_deferred_elements** maximum number of deferred elements before skipping starts. Each bucket may defer updates, but once the total number of deferred updates summed across all buckets reaches this value then all buckets will skip subsequent updates.
- **drain_until** time at which any remaining deferred elements must be skipped and the iterator stops. Once the wrapped iterator has been exhausted, this iterator will drain deferred elements from its buckets until either all buckets have drained or this time is reached.

next()

class swift.obj.updater.**ObjectUpdater**(*conf*, *logger=None*)

Bases: swift.common.daemon.Daemon

Update object information in container listings.

get_container_ring()

Get the container ring. Load it, if it hasnt been yet.

object_sweep(*device*)

If there are async pendings on the device, walk each one and update.

Parameters **device** path to device

object_update(*node*, *part*, *op*, *obj*, *headers_out*)

Perform the object update to the container

Parameters

- **node** node dictionary from the container ring
- **part** partition that holds the container
- **op** operation performed (ex: PUT or DELETE)
- **obj** object name being updated
- **headers_out** headers to send with the update

Returns a tuple of (success, node_id, redirect) where success is True if the update succeeded, node_id is the_id of the node updated and redirect is either None or a tuple of (a path, a timestamp string).

process_object_update(*update_path*, *device*, *policy*, *update*, ***kwargs*)

Process the object information to be updated and update.

Parameters

- **update_path** path to pickled object update file
- **device** path to device
- **policy** storage policy of object update
- **update** the un-pickled update data
- **kwargs** un-used keys from update_ctx

run_forever(**args*, ***kwargs*)

Run the updater continuously.

run_once(**args*, ***kwargs*)

Run the updater once.

class swift.obj.updater.**RateLimiterBucket**(*max_updates_per_second*)

Bases: swift.common.utils.EventletRateLimiter

Extends EventletRateLimiter to also maintain a deque of items that have been deferred due to rate-limiting, and to provide a comparator for sorting instanced by readiness.

class swift.obj.updater.**SweepStats**(*errors=0*, *failures=0*, *quarantines=0*, *successes=0*, *unlinks=0*, *redirects=0*, *skips=0*, *deferrals=0*, *drains=0*)

Bases: object

Stats bucket for an update sweep

A measure of the rate at which updates are being rate-limited is:

```
deferrals / (deferrals + successes + failures - drains)
```

A measure of the rate at which updates are not being sent during a sweep is:

```
skips / (skips + successes + failures)
```

copy()

reset()

since(*other*)

swift.obj.updater.**random**() → x in the interval [0, 1).

swift.obj.updater.**split_update_path**(*update*)

Split the account and container parts out of the async update data.

N.B. updates to shards set the container_path key while the account and container keys are always the root.

9.7 Misc

9.7.1 ACLs

swift.common.middleware.acl.**acls_from_account_info**(*info*)

Extract the account ACLs from the given account_info, and return the ACLs.

Parameters **info** a dict of the form returned by get_account_info

Returns None (no ACL system metadata is set), or a dict of the form:: {admin: [], read-write: [], read-only: []}

Raises **ValueError** on a syntactically invalid header

swift.common.middleware.acl.**clean_acl**(*name*, *value*)

Returns a cleaned ACL header value, validating that it meets the formatting requirements for standard Swift ACL strings.

The ACL format is:

```
[item[,item...]]
```

Each item can be a group name to give access to or a referrer designation to grant or deny based on the HTTP Referer header.

The referrer designation format is:

```
.r:[-]value
```

The `.r` can also be `.ref`, `.referer`, or `.referrer`; though it will be shortened to just `.r` for decreased character count usage.

The value can be * to specify any referrer host is allowed access, a specific host name like `www.example.com`, or if it has a leading period `.` or leading `*.` it is a domain name specification, like `.example.com` or `*.example.com`. The leading minus sign `-` indicates referrer hosts that should be denied access.

Referrer access is applied in the order they are specified. For example, .r:.example.com,.r:-thief.example.com would allow all hosts ending with .example.com except for the specific host thief.example.com.

Example valid ACLs:

```
.r:*
.r:*,.r:-.thief.com
.r:*,.r:.example.com,.r:-thief.example.com
.r:*,.r:-.thief.com,bobs_account,sues_account:sue
bobs_account,sues_account:sue
```

Example invalid ACLs:

```
.r:
.r:-
```

By default, allowing read access via .r will not allow listing objects in the container just retrieving objects from the container. To turn on listings, use the .rlistings directive.

Also, .r designations arent allowed in headers whose names include the word write.

ACLs that are messy will be cleaned up. Examples:

Original	Cleaned
`bob, sue`	`bob,sue`
`bob , sue`	`bob,sue`
`bob,,,sue`	`bob,sue`
`.referrer : *`	`.r:*`
`.ref:*.example.com`	`.r:.example.com`
`.r:*, .rlistings`	`.r:*,.rlistings`

Parameters

- **name** The name of the header being cleaned, such as X-Container-Read or X-Container-Write.
- **value** The value of the header being cleaned.

Returns The value, cleaned of extraneous formatting.

Raises **ValueError** If the value does not meet the ACL formatting requirements; the error message will indicate why.

swift.common.middleware.acl.**format_acl**(*version=1*, ***kwargs*)

Compatibility wrapper to help migrate ACL syntax from version 1 to 2. Delegates to the appropriate version-specific format_acl method, defaulting to version 1 for backward compatibility.

Parameters **kwargs** keyword args appropriate for the selected ACL syntax version (see *format_acl_v1()* or *format_acl_v2()*)

swift.common.middleware.acl.**format_acl_v1**(*groups=None*, *referrers=None*, *header_name=None*)

Returns a standard Swift ACL string for the given inputs.

Caller is responsible for ensuring that :referrers: parameter is only given if the ACL is being generated for X-Container-Read. (X-Container-Write and the account ACL headers dont support referrers.)

Parameters

- **groups** a list of groups (and/or members in most auth systems) to grant access
- **referrers** a list of referrer designations (without the leading .r:)
- **header_name** (optional) header name of the ACL were preparing, for clean_acl; if None, returned ACL wont be cleaned

Returns a Swift ACL string for use in X-Container-{Read,Write}, X-Account-Access-Control, etc.

swift.common.middleware.acl.**format_acl_v2**(*acl_dict*)

Returns a version-2 Swift ACL JSON string.

HTTP headers for Version 2 ACLs have the following form: Header-Name: {arbitrary:json,encoded:string}

JSON will be forced ASCII (containing six-char uNNNN sequences rather than UTF-8; UTF-8 is valid JSON but clients vary in their support for UTF-8 headers), and without extraneous whitespace.

Advantages over V1: forward compatibility (new keys dont cause parsing exceptions); Unicode support; no reserved words (you can have a user named .rlistings if you want).

Parameters **acl_dict** dict of arbitrary data to put in the ACL; see specific auth systems such as tempauth for supported values

Returns a JSON string which encodes the ACL

swift.common.middleware.acl.**parse_acl**(**args*, ***kwargs*)

Compatibility wrapper to help migrate ACL syntax from version 1 to 2. Delegates to the appropriate version-specific parse_acl method, attempting to determine the version from the types of args/kwargs.

Parameters

- **args** positional args for the selected ACL syntax version
- **kwargs** keyword args for the selected ACL syntax version (see *parse_acl_v1()* or *parse_acl_v2()*)

Returns the return value of *parse_acl_v1()* or *parse_acl_v2()*

swift.common.middleware.acl.**parse_acl_v1**(*acl_string*)

Parses a standard Swift ACL string into a referrers list and groups list.

See *clean_acl()* for documentation of the standard Swift ACL format.

Parameters **acl_string** The standard Swift ACL string to parse.

Returns A tuple of (referrers, groups) where referrers is a list of referrer designations (without the leading .r:) and groups is a list of groups to allow access.

swift.common.middleware.acl.**parse_acl_v2**(*data*)

Parses a version-2 Swift ACL string and returns a dict of ACL info.

Parameters **data** string containing the ACL data in JSON format

Returns A dict (possibly empty) containing ACL info, e.g.: {groups: [], referrers: []}

Returns None if data is None, is not valid JSON or does not parse as a dict

Returns empty dictionary if data is an empty string

swift.common.middleware.acl.**referrer_allowed**(*referrer*, *referrer_acl*)

Returns True if the referrer should be allowed based on the referrer_acl list (as returned by *parse_acl()*).

See *clean_acl()* for documentation of the standard Swift ACL format.

Parameters

- **referrer** The value of the HTTP Referer header.
- **referrer_acl** The list of referrer designations as returned by *parse_acl()*.

Returns True if the referrer should be allowed; False if not.

9.7.2 Buffered HTTP

Monkey Patch httplib.HTTPResponse to buffer reads of headers. This can improve performance when making large numbers of small HTTP requests. This module also provides helper functions to make HTTP connections using BufferedHTTPResponse.

Warning: If you use this, be sure that the libraries you are using do not access the socket directly (xmlrpclib, Im looking at you :/), and instead make all calls through httplib.

class swift.common.bufferedhttp.**BufferedHTTPConnection**(*host*, *port=None*, *timeout=<object object>*, *source_address=None*)

Bases: eventlet.green.http.client.HTTPConnection

HTTPConnection class that uses BufferedHTTPResponse

connect()

Connect to the host and port specified in __init__.

getresponse()

Get the response from the server.

If the HTTPConnection is in the correct state, returns an instance of HTTPResponse or of whatever object is returned by the response_class variable.

If a request has not been sent or if a previous response has not be handled, ResponseNotReady is raised. If the HTTP response indicates that the connection should be closed, then it will be closed before the response is returned. When the connection is closed, the underlying socket is closed.

putheader(*header*, *value*)

Send a request header line to the server.

For example: h.putheader(Accept, text/html)

putrequest(*method*, *url*, *skip_host=0*, *skip_accept_encoding=0*)

Send a request to the server.

Parameters

- **method** specifies an HTTP request method, e.g. GET.
- **url** specifies the object being requested, e.g. /index.html.
- **skip_host** if True does not add automatically a Host: header
- **skip_accept_encoding** if True does not add automatically an Accept-Encoding: header

response_class

alias of *swift.common.bufferedhttp.BufferedHTTPResponse*

class swift.common.bufferedhttp.**BufferedHTTPResponse**(*sock*, *debuglevel=0*, *strict=0*, *method=None*)

Bases: eventlet.green.http.client.HTTPResponse

HTTPResponse class that buffers reading of headers

close()

Flush and close the IO object.

This method has no effect if the file is already closed.

nuke_from_orbit()

Terminate the socket with extreme prejudice.

Closes the underlying socket regardless of whether or not anyone else has references to it. Use this when you are certain that nobody else you care about has a reference to this socket.

read(*amt=None*)

Read and return up to n bytes.

If the argument is omitted, None, or negative, reads and returns all data until EOF.

If the argument is positive, and the underlying raw stream is not interactive, multiple raw reads may be issued to satisfy the byte count (unless EOF is reached first). But for interactive raw streams (as well as sockets and pipes), at most one raw read will be issued, and a short result does not imply that EOF is imminent.

Returns an empty bytes object on EOF.

Returns None if the underlying raw stream was open in non-blocking mode and no data is available at the moment.

readline(*size=1024*)

Read and return a line from the stream.

If size is specified, at most size bytes will be read.

The line terminator is always bn for binary files; for text files, the newlines argument to open can be used to select the line terminator(s) recognized.

swift.common.bufferedhttp.**http_connect**(*ipaddr*, *port*, *device*, *partition*, *method*, *path*, *headers=None*, *query_string=None*, *ssl=False*)

Helper function to create an HTTPConnection object. If ssl is set True, HTTPSConnection will be used. However, if ssl=False, BufferedHTTPConnection will be used, which is buffered for backend Swift services.

Parameters

- **ipaddr** IPv4 address to connect to
- **port** port to connect to
- **device** device of the node to query
- **partition** partition on the device
- **method** HTTP method to request (GET, PUT, POST, etc.)
- **path** request path
- **headers** dictionary of headers
- **query_string** request query string
- **ssl** set True if SSL should be used (default: False)

Returns HTTPConnection object

swift.common.bufferedhttp.**http_connect_raw**(*ipaddr*, *port*, *method*, *path*, *headers=None*, *query_string=None*, *ssl=False*)

Helper function to create an HTTPConnection object. If ssl is set True, HTTPSConnection will be used. However, if ssl=False, BufferedHTTPConnection will be used, which is buffered for backend Swift services.

Parameters

- **ipaddr** IPv4 address to connect to
- **port** port to connect to
- **method** HTTP method to request (GET, PUT, POST, etc.)
- **path** request path
- **headers** dictionary of headers
- **query_string** request query string
- **ssl** set True if SSL should be used (default: False)

Returns HTTPConnection object

9.7.3 Constraints

swift.common.constraints.**check_account_format**(*req*, *name*, *, *target_type='Account'*)

Validate that the header contains valid account or container name.

Parameters

- **req** HTTP request object
- **name** header value to validate

- **target_type** which header is being validated (Account or Container)

Returns A properly encoded account name or container name

Raises **HTTPPreconditionFailed** if account header is not well formatted.

swift.common.constraints.**check_container_format**(*req*, *name*, *, *target_type='Container'*)

Validate that the header contains valid account or container name.

Parameters

- **req** HTTP request object
- **name** header value to validate
- **target_type** which header is being validated (Account or Container)

Returns A properly encoded account name or container name

Raises **HTTPPreconditionFailed** if account header is not well formatted.

swift.common.constraints.**check_delete_headers**(*request*)

Check that x-delete-after and x-delete-at headers have valid values. Values should be positive integers and correspond to a time greater than the request timestamp.

If the x-delete-after header is found then its value is used to compute an x-delete-at value which takes precedence over any existing x-delete-at header.

Parameters **request** the swob request object

Raises HTTPBadRequest in case of invalid values

Returns the swob request object

swift.common.constraints.**check_dir**(*root*, *drive*)

Verify that the path to the device is a directory and is a lesser constraint that is enforced when a full mount_check isnt possible with, for instance, a VM using loopback or partitions.

Parameters

- **root** base path where the dir is
- **drive** drive name to be checked

Returns full path to the device

Raises **ValueError** if drive fails to validate

swift.common.constraints.**check_drive**(*root*, *drive*, *mount_check*)

Validate the path given by root and drive is a valid existing directory.

Parameters

- **root** base path where the devices are mounted
- **drive** drive name to be checked
- **mount_check** additionally require path is mounted

Returns full path to the device

Raises **ValueError** if drive fails to validate

swift.common.constraints.**check_float**(*string*)

Helper function for checking if a string can be converted to a float.

Parameters **string** string to be verified as a float

Returns True if the string can be converted to a float, False otherwise

swift.common.constraints.**check_metadata**(*req*, *target_type*)

Check metadata sent in the request headers. This should only check that the metadata in the request given is valid. Checks against account/container overall metadata should be forwarded on to its respective server to be checked.

Parameters

- **req** request object
- **target_type** str: one of: object, container, or account: indicates which type the target storage for the metadata is

Returns HTTPBadRequest with bad metadata otherwise None

swift.common.constraints.**check_mount**(*root*, *drive*)

Verify that the path to the device is a mount point and mounted. This allows us to fast fail on drives that have been unmounted because of issues, and also prevents us for accidentally filling up the root partition.

Parameters

- **root** base path where the devices are mounted
- **drive** drive name to be checked

Returns full path to the device

Raises **ValueError** if drive fails to validate

swift.common.constraints.**check_name_format**(*req*, *name*, *target_type*)

Validate that the header contains valid account or container name.

Parameters

- **req** HTTP request object
- **name** header value to validate
- **target_type** which header is being validated (Account or Container)

Returns A properly encoded account name or container name

Raises **HTTPPreconditionFailed** if account header is not well formatted.

swift.common.constraints.**check_object_creation**(*req*, *object_name*)

Check to ensure that everything is alright about an object to be created.

Parameters

- **req** HTTP request object
- **object_name** name of object to be created

Returns HTTPRequestEntityTooLarge the object is too large

Returns HTTPLengthRequired missing content-length header and not a chunked request

Returns HTTPBadRequest missing or bad content-type header, or bad metadata

Returns HTTPNotImplemented unsupported transfer-encoding header value

swift.common.constraints.**check_utf8**(*string*, *internal=False*)

Validate if a string is valid UTF-8 str or unicode and that it does not contain any reserved characters.

Parameters

- **string** string to be validated
- **internal** boolean, allows reserved characters if True

Returns True if the string is valid utf-8 str or unicode and contains no null characters, False otherwise

swift.common.constraints.**reload_constraints**()

Parse SWIFT_CONF_FILE and reset module level global constraint attrs, populating OVERRIDE_CONSTRAINTS AND EFFECTIVE_CONSTRAINTS along the way.

swift.common.constraints.**valid_api_version**(*version*)

Checks if the requested version is valid.

Currently Swift only supports v1 and v1.0.

swift.common.constraints.**valid_timestamp**(*request*)

Helper function to extract a timestamp from requests that require one.

Parameters **request** the swob request object

Returns a valid Timestamp instance

Raises **HTTPBadRequest** on missing or invalid X-Timestamp

9.7.4 Container Sync Realms

class swift.common.container_sync_realms.**ContainerSyncRealms**(*conf_path*, *logger*)

Bases: object

Loads and parses the container-sync-realms.conf, occasionally checking the files mtime to see if it needs to be reloaded.

clusters(*realm*)

Returns a list of clusters for the realm.

endpoint(*realm*, *cluster*)

Returns the endpoint for the cluster in the realm.

get_sig(*request_method*, *path*, *x_timestamp*, *nonce*, *realm_key*, *user_key*)

Returns the hexdigest string of the HMAC-SHA1 (RFC 2104) for the information given.

Parameters

- **request_method** HTTP method of the request.
- **path** The path to the resource (url-encoded).

- **x_timestamp** The X-Timestamp header value for the request.
- **nonce** A unique value for the request.
- **realm_key** Shared secret at the cluster operator level.
- **user_key** Shared secret at the users container level.

Returns hexdigest str of the HMAC-SHA1 for the request.

key(*realm*)

Returns the key for the realm.

key2(*realm*)

Returns the key2 for the realm.

realms()

Returns a list of realms.

reload()

Forces a reload of the conf file.

9.7.5 Digest

swift.common.digest.**extract_digest_and_algorithm**(*value*)

Returns a tuple of (digest_algorithm, hex_encoded_digest) from a client-provided string of the form:

```
<hex-encoded digest>
```

or:

```
<algorithm>:<base64-encoded digest>
```

Note that hex-encoded strings must use one of sha1, sha256, or sha512.

Raises ValueError on parse failures

swift.common.digest.**get_allowed_digests**(*conf_digests*, *logger=None*)

Pulls out allowed_digests from the supplied conf. Then compares them with the list of supported and deprecated digests and returns whatever remain.

When something is unsupported or deprecated itll log a warning.

Parameters

- **conf_digests** iterable of allowed digests. If empty, defaults to DEFAULT_ALLOWED_DIGESTS.
- **logger** optional logger; if provided, use it issue deprecation warnings

Returns A set of allowed digests that are supported and a set of deprecated digests.

Raises ValueError, if there are no digests left to return.

swift.common.digest.**get_hmac**(*request_method*, *path*, *expires*, *key*, *digest='sha1'*, *ip_range=None*)

Returns the hexdigest string of the HMAC (see RFC 2104) for the request.

Parameters

- **request_method** Request method to allow.
- **path** The path to the resource to allow access to.
- **expires** Unix timestamp as an int for when the URL expires.
- **key** HMAC shared secret.
- **digest** constructor or the string name for the digest to use in calculating the HMAC Defaults to SHA1
- **ip_range** The ip range from which the resource is allowed to be accessed. We need to put the ip_range as the first argument to hmac to avoid manipulation of the path due to newlines being valid in paths e.g. /v1/a/c/on127.0.0.1

Returns hexdigest str of the HMAC for the request using the specified digest algorithm.

9.7.6 Direct Client

Internal client library for making calls directly to the servers rather than through the proxy.

exception swift.common.direct_client.**DirectClientException**(*stype*, *method*, *node*, *part*, *path*, *resp*, *host=None*)

Bases: *swift.common.exceptions.ClientException*

exception swift.common.direct_client.**DirectClientReconException**(*method*, *node*, *path*, *resp*)

Bases: *swift.common.exceptions.ClientException*

swift.common.direct_client.**direct_delete_account**(*node*, *part*, *account*, *conn_timeout=5*, *response_timeout=15*, *headers=None*)

swift.common.direct_client.**direct_delete_container**(*node*, *part*, *account*, *container*, *conn_timeout=5*, *response_timeout=15*, *headers=None*)

Delete container directly from the container server.

Parameters

- **node** node dictionary from the ring
- **part** partition the container is on
- **account** account name
- **container** container name
- **conn_timeout** timeout in seconds for establishing the connection
- **response_timeout** timeout in seconds for getting the response
- **headers** dict to be passed into HTTPConnection headers

Raises *ClientException* HTTP DELETE request failed

swift.common.direct_client.**direct_delete_container_object**(*node*, *part*, *account*, *container*, *obj*, *conn_timeout=5*, *response_timeout=15*, *headers=None*)

swift.common.direct_client.**direct_delete_object**(*node*, *part*, *account*, *container*, *obj*, *conn_timeout=5*, *response_timeout=15*, *headers=None*)

Delete object directly from the object server.

Parameters

- **node** node dictionary from the ring
- **part** partition the container is on
- **account** account name
- **container** container name
- **obj** object name
- **conn_timeout** timeout in seconds for establishing the connection
- **response_timeout** timeout in seconds for getting the response

Raises *ClientException* HTTP DELETE request failed

swift.common.direct_client.**direct_get_account**(*node*, *part*, *account*, *marker=None*, *limit=None*, *prefix=None*, *delimiter=None*, *conn_timeout=5*, *response_timeout=15*, *end_marker=None*, *reverse=None*, *headers=None*)

Get listings directly from the account server.

Parameters

- **node** node dictionary from the ring
- **part** partition the account is on
- **account** account name
- **marker** marker query
- **limit** query limit
- **prefix** prefix query
- **delimiter** delimiter for the query
- **conn_timeout** timeout in seconds for establishing the connection
- **response_timeout** timeout in seconds for getting the response
- **end_marker** end_marker query
- **reverse** reverse the returned listing

Returns a tuple of (response headers, a list of containers) The response headers will HeaderKeyDict.

swift.common.direct_client.**direct_get_container**(*node*, *part*, *account*, *container*, *marker=None*, *limit=None*, *prefix=None*, *delimiter=None*, *conn_timeout=5*, *response_timeout=15*, *end_marker=None*, *reverse=None*, *headers=None*)

Get container listings directly from the container server.

Parameters

- **node** node dictionary from the ring
- **part** partition the container is on
- **account** account name
- **container** container name
- **marker** marker query
- **limit** query limit
- **prefix** prefix query
- **delimiter** delimiter for the query
- **conn_timeout** timeout in seconds for establishing the connection
- **response_timeout** timeout in seconds for getting the response
- **end_marker** end_marker query
- **reverse** reverse the returned listing
- **headers** headers to be included in the request

Returns a tuple of (response headers, a list of objects) The response headers will be a HeaderKeyDict.

swift.common.direct_client.**direct_get_object**(*node*, *part*, *account*, *container*, *obj*, *conn_timeout=5*, *response_timeout=15*, *resp_chunk_size=None*, *headers=None*)

Get object directly from the object server.

Parameters

- **node** node dictionary from the ring
- **part** partition the container is on
- **account** account name
- **container** container name
- **obj** object name
- **conn_timeout** timeout in seconds for establishing the connection
- **response_timeout** timeout in seconds for getting the response
- **resp_chunk_size** if defined, chunk size of data to read.
- **headers** dict to be passed into HTTPConnection headers

Returns a tuple of (response headers, the objects contents) The response headers will be a HeaderKeyDict.

Raises *ClientException* HTTP GET request failed

swift.common.direct_client.**direct_get_recon**(*node*, *recon_command*, *conn_timeout=5*, *response_timeout=15*, *headers=None*)

Get recon json directly from the storage server.

Parameters

- **node** node dictionary from the ring
- **recon_command** recon string (post /recon/)
- **conn_timeout** timeout in seconds for establishing the connection
- **response_timeout** timeout in seconds for getting the response
- **headers** dict to be passed into HTTPConnection headers

Returns deserialized json response

Raises *DirectClientReconException* HTTP GET request failed

swift.common.direct_client.**direct_get_suffix_hashes**(*node*, *part*, *suffixes*, *conn_timeout=5*, *response_timeout=15*, *headers=None*)

Get suffix hashes directly from the object server.

Note that unlike other `direct_client` functions, this one defaults to using the replication network to make requests.

Parameters

- **node** node dictionary from the ring
- **part** partition the container is on
- **conn_timeout** timeout in seconds for establishing the connection
- **response_timeout** timeout in seconds for getting the response
- **headers** dict to be passed into HTTPConnection headers

Returns dict of suffix hashes

Raises *ClientException* HTTP REPLICATE request failed

swift.common.direct_client.**direct_head_container**(*node*, *part*, *account*, *container*, *conn_timeout=5*, *response_timeout=15*, *headers=None*)

Request container information directly from the container server.

Parameters

- **node** node dictionary from the ring
- **part** partition the container is on
- **account** account name

- **container** container name
- **conn_timeout** timeout in seconds for establishing the connection
- **response_timeout** timeout in seconds for getting the response

Returns a dict containing the responses headers in a HeaderKeyDict

Raises *ClientException* HTTP HEAD request failed

swift.common.direct_client.**direct_head_object**(*node*, *part*, *account*, *container*, *obj*, *conn_timeout=5*, *response_timeout=15*, *headers=None*)

Request object information directly from the object server.

Parameters

- **node** node dictionary from the ring
- **part** partition the container is on
- **account** account name
- **container** container name
- **obj** object name
- **conn_timeout** timeout in seconds for establishing the connection
- **response_timeout** timeout in seconds for getting the response
- **headers** dict to be passed into HTTPConnection headers

Returns a dict containing the responses headers in a HeaderKeyDict

Raises *ClientException* HTTP HEAD request failed

swift.common.direct_client.**direct_post_object**(*node*, *part*, *account*, *container*, *name*, *headers*, *conn_timeout=5*, *response_timeout=15*)

Direct update to object metadata on object server.

Parameters

- **node** node dictionary from the ring
- **part** partition the container is on
- **account** account name
- **container** container name
- **name** object name
- **headers** headers to store as metadata
- **conn_timeout** timeout in seconds for establishing the connection
- **response_timeout** timeout in seconds for getting the response

Raises *ClientException* HTTP POST request failed

swift.common.direct_client.**direct_put_container**(*node*, *part*, *account*, *container*, *conn_timeout=5*, *response_timeout=15*, *headers=None*, *contents=None*, *content_length=None*, *chunk_size=65535*)

Make a PUT request to a container server.

Parameters

- **node** node dictionary from the ring
- **part** partition the container is on
- **account** account name
- **container** container name
- **conn_timeout** timeout in seconds for establishing the connection
- **response_timeout** timeout in seconds for getting the response
- **headers** additional headers to include in the request
- **contents** an iterable or string to send in request body (optional)
- **content_length** value to send as content-length header (optional)
- **chunk_size** chunk size of data to send (optional)

Raises *ClientException* HTTP PUT request failed

swift.common.direct_client.**direct_put_container_object**(*node*, *part*, *account*, *container*, *obj*, *conn_timeout=5*, *response_timeout=15*, *headers=None*)

swift.common.direct_client.**direct_put_object**(*node*, *part*, *account*, *container*, *name*, *contents*, *content_length=None*, *etag=None*, *content_type=None*, *headers=None*, *conn_timeout=5*, *response_timeout=15*, *chunk_size=65535*)

Put object directly from the object server.

Parameters

- **node** node dictionary from the ring
- **part** partition the container is on
- **account** account name
- **container** container name
- **name** object name
- **contents** an iterable or string to read object data from
- **content_length** value to send as content-length header
- **etag** etag of contents
- **content_type** value to send as content-type header

- **headers** additional headers to include in the request
- **conn_timeout** timeout in seconds for establishing the connection
- **response_timeout** timeout in seconds for getting the response
- **chunk_size** if defined, chunk size of data to send.

Returns etag from the server response

Raises *ClientException* HTTP PUT request failed

swift.common.direct_client.**gen_headers**(*hdrs_in=None*, *add_ts=True*)

Get the headers ready for a request. All requests should have a User-Agent string, but if one is passed in dont over-write it. Not all requests will need an X-Timestamp, but if one is passed in do not over-write it.

Parameters

- **headers** dict or None, base for HTTP headers
- **add_ts** boolean, should be True for any unsafe HTTP request

Returns HeaderKeyDict based on headers and ready for the request

swift.common.direct_client.**retry**(*func*, **args*, ***kwargs*)

Helper function to retry a given function a number of times.

Parameters

- **func** callable to be called
- **retries** number of retries
- **error_log** logger for errors
- **args** arguments to send to func
- **kwargs** keyward arguments to send to func (if retries or error_log are sent, they will be deleted from kwargs before sending on to func)

Returns result of func

Raises *ClientException* all retries failed

9.7.7 Exceptions

exception swift.common.exceptions.**APIVersionError**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**ChunkReadError**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**ChunkReadTimeout**(*seconds=None*, *exception=None*)

Bases: eventlet.timeout.Timeout

exception swift.common.exceptions.**ChunkWriteTimeout**(*seconds=None*, *exception=None*)

Bases: eventlet.timeout.Timeout

exception swift.common.exceptions.**ClientException**(*msg*, *http_scheme=''*, *http_host=''*, *http_port=''*, *http_path=''*, *http_query=''*, *http_status=None*, *http_reason=''*, *http_device=''*, *http_response_content=''*, *http_headers=None*)

Bases: Exception

exception swift.common.exceptions.**ConnectionTimeout**(*seconds=None*, *exception=None*)

Bases: eventlet.timeout.Timeout

exception swift.common.exceptions.**DatabaseAuditorException**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**DeviceUnavailable**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**DiskFileBadMetadataChecksum**

Bases: *swift.common.exceptions.DiskFileError*

exception swift.common.exceptions.**DiskFileCollision**

Bases: *swift.common.exceptions.DiskFileError*

exception swift.common.exceptions.**DiskFileDeleted**(*metadata=None*)

Bases: *swift.common.exceptions.DiskFileNotExist*

exception swift.common.exceptions.**DiskFileDeviceUnavailable**

Bases: *swift.common.exceptions.DiskFileError*

exception swift.common.exceptions.**DiskFileError**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**DiskFileExpired**(*metadata=None*)

Bases: *swift.common.exceptions.DiskFileDeleted*

exception swift.common.exceptions.**DiskFileNoSpace**

Bases: *swift.common.exceptions.DiskFileError*

exception swift.common.exceptions.**DiskFileNotExist**

Bases: *swift.common.exceptions.DiskFileError*

exception swift.common.exceptions.**DiskFileNotOpen**

Bases: *swift.common.exceptions.DiskFileError*

exception swift.common.exceptions.**DiskFileQuarantined**

Bases: *swift.common.exceptions.DiskFileError*

exception swift.common.exceptions.**DiskFileXattrNotSupported**

Bases: *swift.common.exceptions.DiskFileError*

exception swift.common.exceptions.**DriveNotMounted**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**DuplicateDeviceError**

Bases: *swift.common.exceptions.RingBuilderError*

exception swift.common.exceptions.**EmptyRingError**

Bases: *swift.common.exceptions.RingBuilderError*

exception swift.common.exceptions.**EncryptionException**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**FileNotFoundError**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**FooterNotSupported**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**InsufficientStorage**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**InvalidAccountInfo**

Bases: *swift.common.exceptions.DatabaseAuditorException*

exception swift.common.exceptions.**InvalidPidFileException**

Bases: Exception

exception swift.common.exceptions.**InvalidTimestamp**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**LinkIterError**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**ListingIterError**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**ListingIterNotAuthorized**(*aresp*)

Bases: *swift.common.exceptions.ListingIterError*

exception swift.common.exceptions.**ListingIterNotFound**

Bases: *swift.common.exceptions.ListingIterError*

exception swift.common.exceptions.**LockTimeout**(*seconds=None*, *msg=None*)

Bases: *swift.common.exceptions.MessageTimeout*

exception swift.common.exceptions.**MessageTimeout**(*seconds=None*, *msg=None*)

Bases: eventlet.timeout.Timeout

exception swift.common.exceptions.**MimeInvalid**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**MultiphasePUTNotSupported**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**PartitionLockTimeout**(*seconds=None*, *msg=None*)

Bases: *swift.common.exceptions.LockTimeout*

exception swift.common.exceptions.**PathNotDir**

Bases: OSError

exception swift.common.exceptions.**PermissionError**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**PutterConnectError**(*status=None*)

Bases: Exception

exception swift.common.exceptions.**QuarantineRequest**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**RangeAlreadyComplete**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**ReplicationException**

Bases: Exception

exception swift.common.exceptions.**ReplicationLockTimeout**(*seconds=None*, *msg=None*)

Bases: *swift.common.exceptions.LockTimeout*

exception swift.common.exceptions.**ResponseTimeout**(*seconds=None*, *exception=None*)

Bases: eventlet.timeout.Timeout

exception swift.common.exceptions.**RingBuilderError**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**RingLoadError**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**RingValidationError**

Bases: *swift.common.exceptions.RingBuilderError*

exception swift.common.exceptions.**SegmentError**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**ShortReadError**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**SuffixSyncError**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**SwiftException**

Bases: Exception

exception swift.common.exceptions.**UnPicklingError**

Bases: *swift.common.exceptions.SwiftException*

exception swift.common.exceptions.**UnknownSecretIdError**

Bases: *swift.common.exceptions.EncryptionException*

9.7.8 Internal Client

class swift.common.internal_client.**CompressingFileReader**(*file_obj*, *compresslevel=9*, *chunk_size=4096*)

Bases: object

Wrapper for file object to compress object while reading.

Can be used to wrap file objects passed to InternalClient.upload_object().

Used in testing of InternalClient.

Parameters

- **file_obj** File object to wrap.
- **compresslevel** Compression level, defaults to 9.
- **chunk_size** Size of chunks read when iterating using object, defaults to 4096.

next()

read(**a*, ***kw*)

Reads a chunk from the file object.

Params are passed directly to the underlying file objects read().

Returns Compressed chunk from file object.

seek(*offset*, *whence=0*)

set_initial_state()

Sets the object to the state needed for the first read.

class swift.common.internal_client.**InternalClient**(*conf_path*, *user_agent*, *request_tries*, *allow_modify_pipeline=False*, *use_replication_network=False*, *global_conf=None*, *app=None*)

Bases: object

An internal client that uses a swift proxy app to make requests to Swift.

This client will exponentially slow down for retries.

Parameters

- **conf_path** Full path to proxy config.
- **user_agent** User agent to be sent to requests to Swift.
- **request_tries** Number of tries before InternalClient.make_request() gives up.
- **global_conf** a dict of options to update the loaded proxy config. Options in global_conf will override those in conf_path except where the conf_path option is preceded by set.
- **app** Optionally provide a WSGI app for the internal client to use.

container_exists(*account*, *container*)

Checks to see if a container exists.

Parameters

- **account** The containers account.
- **container** Container to check.

Raises

- *UnexpectedResponse* Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

Returns True if container exists, false otherwise.

`create_account`(*account*)

Creates an account.

Parameters **account** Account to create.

Raises

- *UnexpectedResponse* Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

`create_container`(*account*, *container*, *headers=None*, *acceptable_statuses=(2,)*)

Creates container.

Parameters

- **account** The containers account.
- **container** Container to create.
- **headers** Defaults to empty dict.
- **acceptable_statuses** List of status for valid responses, defaults to (2,).

Raises

- *UnexpectedResponse* Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

`delete_account`(*account*, *acceptable_statuses=(2, 404)*)

Deletes an account.

Parameters

- **account** Account to delete.
- **acceptable_statuses** List of status for valid responses, defaults to (2, HTTP_NOT_FOUND).

Raises

- *UnexpectedResponse* Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

`delete_container`(*account*, *container*, *headers=None*, *acceptable_statuses=(2, 404)*)

Deletes a container.

Parameters

- **account** The containers account.
- **container** Container to delete.
- **acceptable_statuses** List of status for valid responses, defaults to (2, HTTP_NOT_FOUND).

Raises

- *UnexpectedResponse* Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

delete_object(*account*, *container*, *obj*, *acceptable_statuses=(2, 404)*, *headers=None*)

Deletes an object.

Parameters

- **account** The objects account.
- **container** The objects container.
- **obj** The object.
- **acceptable_statuses** List of status for valid responses, defaults to (2, HTTP_NOT_FOUND).
- **headers** extra headers to send with request

Raises

- *UnexpectedResponse* Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

get_account_info(*account*, *acceptable_statuses=(2, 404)*)

Returns (container_count, object_count) for an account.

Parameters

- **account** Account on which to get the information.
- **acceptable_statuses** List of status for valid responses, defaults to (2, HTTP_NOT_FOUND).

Raises

- *UnexpectedResponse* Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

get_account_metadata(*account*, *metadata_prefix=''*, *acceptable_statuses=(2,)*, *params=None*)

Gets account metadata.

Parameters

- **account** Account on which to get the metadata.
- **metadata_prefix** Used to filter values from the headers returned. Will strip that prefix from the keys in the dict returned. Defaults to .
- **acceptable_statuses** List of status for valid responses, defaults to (2,).

Returns Returns dict of account metadata. Keys will be lowercase.

Raises

- *UnexpectedResponse* Exception raised when requests fail to get a response with an acceptable status

- **Exception** Exception is raised when code fails in an unexpected way.

get_container_metadata(*account*, *container*, *metadata_prefix=''*, *acceptable_statuses=(2,)*, *params=None*)

Gets container metadata.

Parameters

- **account** The containers account.
- **container** Container to get metadata on.
- **metadata_prefix** Used to filter values from the headers returned. Will strip that prefix from the keys in the dict returned. Defaults to .
- **acceptable_statuses** List of status for valid responses, defaults to (2,).

Returns Returns dict of container metadata. Keys will be lowercase.

Raises

- ***UnexpectedResponse*** Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

get_object(*account*, *container*, *obj*, *headers=None*, *acceptable_statuses=(2,)*, *params=None*)

Gets an object.

Parameters

- **account** The objects account.
- **container** The objects container.
- **obj** The object name.
- **headers** Headers to send with request, defaults to empty dict.
- **acceptable_statuses** List of status for valid responses, defaults to (2,).
- **params** A dict of params to be set in request query string, defaults to None.

Raises

- ***UnexpectedResponse*** Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

Returns A 3-tuple (status, headers, iterator of object body)

get_object_metadata(*account*, *container*, *obj*, *metadata_prefix=''*, *acceptable_statuses=(2,)*, *headers=None*, *params=None*)

Gets object metadata.

Parameters

- **account** The objects account.
- **container** The objects container.
- **obj** The object.

- **metadata_prefix** Used to filter values from the headers returned. Will strip that prefix from the keys in the dict returned. Defaults to .
- **acceptable_statuses** List of status for valid responses, defaults to (2,).
- **headers** extra headers to send with request

Returns Dict of object metadata.

Raises

- *UnexpectedResponse* Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

handle_request(**args*, ***kwargs*)

iter_containers(*account*, *marker=''*, *end_marker=''*, *prefix=''*, *acceptable_statuses=(2, 404)*)

Returns an iterator of containers dicts from an account.

Parameters

- **account** Account on which to do the container listing.
- **marker** Prefix of first desired item, defaults to .
- **end_marker** Last item returned will be less than this, defaults to .
- **prefix** Prefix of containers
- **acceptable_statuses** List of status for valid responses, defaults to (2, HTTP_NOT_FOUND).

Raises

- *UnexpectedResponse* Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

iter_object_lines(*account*, *container*, *obj*, *headers=None*, *acceptable_statuses=(2,)*)

Returns an iterator of object lines from an uncompressed or compressed text object.

Uncompress object as it is read if the objects name ends with .gz.

Parameters

- **account** The objects account.
- **container** The objects container.
- **obj** The object.
- **acceptable_statuses** List of status for valid responses, defaults to (2,).

Raises

- *UnexpectedResponse* Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

iter_objects(*account*, *container*, *marker=''*, *end_marker=''*, *prefix=''*, *acceptable_statuses=(2, 404)*)

Returns an iterator of object dicts from a container.

Parameters

- **account** The containers account.
- **container** Container to iterate objects on.
- **marker** Prefix of first desired item, defaults to .
- **end_marker** Last item returned will be less than this, defaults to .
- **prefix** Prefix of objects
- **acceptable_statuses** List of status for valid responses, defaults to (2, HTTP_NOT_FOUND).

Raises

- *UnexpectedResponse* Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

make_path(*account*, *container=None*, *obj=None*)

Returns a swift path for a request quoting and utf-8 encoding the path parts as need be.

Parameters

- **account** swift account
- **container** container, defaults to None
- **obj** object, defaults to None

Raises **ValueError** Is raised if obj is specified and container is not.

make_request(*method*, *path*, *headers*, *acceptable_statuses*, *body_file=None*, *params=None*)

Makes a request to Swift with retries.

Parameters

- **method** HTTP method of request.
- **path** Path of request.
- **headers** Headers to be sent with request.
- **acceptable_statuses** List of acceptable statuses for request.
- **body_file** Body file to be passed along with request, defaults to None.
- **params** A dict of params to be set in request query string, defaults to None.

Returns Response object on success.

Raises

- *UnexpectedResponse* Exception raised when make_request() fails to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

set_account_metadata(*account*, *metadata*, *metadata_prefix=''*, *acceptable_statuses=(2,)*)

Sets account metadata. A call to this will add to the account metadata and not overwrite all of it with values in the metadata dict. To clear an account metadata value, pass an empty string as the value for the key in the metadata dict.

Parameters

- **account** Account on which to get the metadata.
- **metadata** Dict of metadata to set.
- **metadata_prefix** Prefix used to set metadata values in headers of requests, used to prefix keys in metadata when setting metadata, defaults to .
- **acceptable_statuses** List of status for valid responses, defaults to (2,).

Raises

- ***UnexpectedResponse*** Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

set_container_metadata(*account*, *container*, *metadata*, *metadata_prefix=''*, *acceptable_statuses=(2,)*)

Sets container metadata. A call to this will add to the container metadata and not overwrite all of it with values in the metadata dict. To clear a container metadata value, pass an empty string as the value for the key in the metadata dict.

Parameters

- **account** The containers account.
- **container** Container to set metadata on.
- **metadata** Dict of metadata to set.
- **metadata_prefix** Prefix used to set metadata values in headers of requests, used to prefix keys in metadata when setting metadata, defaults to .
- **acceptable_statuses** List of status for valid responses, defaults to (2,).

Raises

- ***UnexpectedResponse*** Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

set_object_metadata(*account*, *container*, *obj*, *metadata*, *metadata_prefix=''*, *acceptable_statuses=(2,)*)

Sets an objects metadata. The objects metadata will be overwritten by the values in the metadata dict.

Parameters

- **account** The objects account.
- **container** The objects container.
- **obj** The object.

- **metadata** Dict of metadata to set.
- **metadata_prefix** Prefix used to set metadata values in headers of requests, used to prefix keys in metadata when setting metadata, defaults to .
- **acceptable_statuses** List of status for valid responses, defaults to (2,).

Raises

- *UnexpectedResponse* Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

upload_object(*fobj*, *account*, *container*, *obj*, *headers=None*, *acceptable_statuses=(2,)*, *params=None*)

Parameters

- **fobj** File object to read objects content from.
- **account** The objects account.
- **container** The objects container.
- **obj** The object.
- **headers** Headers to send with request, defaults to empty dict.
- **acceptable_statuses** List of acceptable statuses for request.
- **params** A dict of params to be set in request query string, defaults to None.

Raises

- *UnexpectedResponse* Exception raised when requests fail to get a response with an acceptable status
- **Exception** Exception is raised when code fails in an unexpected way.

class swift.common.internal_client.**SimpleClient**(*url=None*, *token=None*, *starting_backoff=1*, *max_backoff=5*, *retries=5*)

Bases: object

Simple client that is used in bin/swift-dispersion-* and container sync

base_request(*method*, *container=None*, *name=None*, *prefix=None*, *headers=None*, *proxy=None*, *contents=None*, *full_listing=None*, *logger=None*, *additional_info=None*, *timeout=None*, *marker=None*)

get_account(**args*, ***kwargs*)

get_container(*container*, ***kwargs*)

put_container(*container*, ***kwargs*)

put_object(*container*, *name*, *contents*, ***kwargs*)

retry_request(*method*, ***kwargs*)

exception swift.common.internal_client.**UnexpectedResponse**(*message*, *resp*)

Bases: Exception

Exception raised on invalid responses to InternalClient.make_request().

Parameters

- **message** Exception message.
- **resp** The unexpected response.

swift.common.internal_client.**delete_object**(*url*, ***kwargs*)

For usage with container sync

swift.common.internal_client.**get_auth**(*url*, *user*, *key*, *auth_version='1.0'*, ***kwargs*)

swift.common.internal_client.**head_object**(*url*, ***kwargs*)

For usage with container sync

swift.common.internal_client.**put_object**(*url*, ***kwargs*)

For usage with container sync

9.7.9 Manager

class swift.common.manager.**Manager**(*servers*, *run_dir='/var/run/swift'*)

Bases: object

Main class for performing commands on groups of servers.

Parameters **servers** list of server names as strings

force_reload(***kwargs*)

alias for reload

get_command(*cmd*)

Find and return the decorated method named like cmd

Parameters **cmd** the command to get, a string, if not found raises UnknownCommandError

kill(***kwargs*)

stop a server (no error if not running)

kill_child_pids(***kwargs*)

kill child pids, optionally servicing accepted connections

classmethod list_commands()

Get all publicly accessible commands

Returns a list of string tuples (cmd, help), the method names who are decorated as commands

no_daemon(***kwargs*)

start a server interactively

no_wait(***kwargs*)

spawn server and return immediately

once(***kwargs*)
start server and run one pass on supporting daemons

reload(***kwargs*)
graceful shutdown then restart on supporting servers

reload_seamless(***kwargs*)
seamlessly re-exec, then shutdown of old listen sockets on supporting servers

restart(***kwargs*)
stops then restarts server

run_command(*cmd*, ***kwargs*)
Find the named command and run it

Parameters **cmd** the command name to run

shutdown(***kwargs*)
allow current requests to finish on supporting servers

start(***kwargs*)
starts a server

status(***kwargs*)
display status of tracked pids for server

stop(***kwargs*)
stops a server

class swift.common.manager.**Server**(*server*, *run_dir='/var/run/swift'*)
Bases: object

Manage operations on a server or group of servers of similar type

Parameters **server** name of server

conf_files(***kwargs*)
Get conf files for this server

Parameters **number** if supplied will only lookup the nth server

Returns list of conf files

get_conf_file_name(*pid_file*)
Translate pid_file to a corresponding conf_file

Parameters **pid_file** a pid_file for this server, a string

Returns the conf_file for this pid_file

get_pid_file_name(*conf_file*)
Translate conf_file to a corresponding pid_file

Parameters **conf_file** an conf_file for this server, a string

Returns the pid_file for this conf_file

get_running_pids(***kwargs*)

Get running pids

Returns a dict mapping pids (ints) to pid_files (paths)

interact(***kwargs*)

wait on spawned procs to terminate

iter_pid_files(***kwargs*)

Generator, yields (pid_file, pids)

kill_child_pids(***kwargs*)

Kill child pids, leaving server overseer to respawn them

Parameters

- **graceful** if True, attempt SIGHUP on supporting servers
- **seamless** if True, attempt SIGUSR1 on supporting servers

Returns a dict mapping pids (ints) to pid_files (paths)

kill_running_pids(***kwargs*)

Kill running pids

Parameters

- **graceful** if True, attempt SIGHUP on supporting servers
- **seamless** if True, attempt SIGUSR1 on supporting servers

Returns a dict mapping pids (ints) to pid_files (paths)

launch(***kwargs*)

Collect conf files and attempt to spawn the processes for this server

pid_files(***kwargs*)

Get pid files for this server

Parameters **number** if supplied will only lookup the nth server

Returns list of pid files

signal_children(*sig*, ***kwargs*)

Send a signal to child pids for this server

Parameters **sig** signal to send

Returns a dict mapping pids (ints) to pid_files (paths)

signal_pids(*sig*, ***kwargs*)

Send a signal to pids for this server

Parameters **sig** signal to send

Returns a dict mapping pids (ints) to pid_files (paths)

spawn(*conf_file*, *once=False*, *wait=True*, *daemon=True*, *additional_args=None*, ***kwargs*)

Launch a subprocess for this server.

Parameters

- **conf_file** path to conf_file to use as first arg
- **once** boolean, add once argument to command
- **wait** boolean, if true capture stdout with a pipe
- **daemon** boolean, if false ask server to log to console
- **additional_args** list of additional arguments to pass on the command line

Returns the pid of the spawned process

status(*pids=None*, ***kwargs*)

Display status of server

Parameters

- **pids** if not supplied pids will be populated automatically
- **number** if supplied will only lookup the nth server

Returns 1 if server is not running, 0 otherwise

stop(***kwargs*)

Send stop signals to pids for this server

Returns a dict mapping pids (ints) to pid_files (paths)

wait(***kwargs*)

wait on spawned procs to start

exception swift.common.manager.**UnknownCommandError**

Bases: Exception

swift.common.manager.**command**(*func*)

Decorator to declare which methods are accessible as commands, commands always return 1 or 0, where 0 should indicate success.

Parameters **func** function to make public

swift.common.manager.**format_server_name**(*servername*)

Formats server name as swift compatible server names E.g. swift-object-server

Parameters **servername** server name

Returns swift compatible server name and its binary name

swift.common.manager.**kill_group**(*pid*, *sig*)

Send signal to process group

: param pid: process id : param sig: signal to send

swift.common.manager.**safe_kill**(*pid*, *sig*, *name*)

Send signal to process and check process name

: param pid: process id : param sig: signal to send : param name: name to ensure target process

swift.common.manager.**setup_env**()

Try to increase resource limits of the OS. Move PYTHON_EGG_CACHE to /tmp

swift.common.manager.**verify_server**(*server*)

Check whether the server is among swift servers or not, and also checks whether the servers binaries are installed or not.

Parameters **server** name of the server

Returns True, when the server name is valid and its binaries are found. False, otherwise.

swift.common.manager.**watch_server_pids**(*server_pids*, *interval=1*, ***kwargs*)

Monitor a collection of server pids yielding back those pids that arent responding to signals.

Parameters **server_pids** a dict, lists of pids [int,] keyed on Server objects

9.7.10 MemCacheD

Why our own memcache client? By Michael Barton

python-memcached doesnt use consistent hashing, so adding or removing a memcache server from the pool invalidates a huge percentage of cached items.

If you keep a pool of python-memcached client objects, each client object has its own connection to every memcached server, only one of which is ever in use. So you wind up with n * m open sockets and almost all of them idle. This client effectively has a pool for each server, so the number of backend connections is hopefully greatly reduced.

python-memcache uses pickle to store things, and there was already a huge stink about Swift using pickles in memcache (http://osvdb.org/show/osvdb/86581). That seemed sort of unfair, since nova and keystone and everyone else use pickles for memcache too, but its hidden behind a standard library. But changing would be a security regression at this point.

Also, pylibmc wouldnt work for us because it needs to use python sockets in order to play nice with eventlet.

Lucid comes with memcached: v1.4.2. Protocol documentation for that version is at:

http://github.com/memcached/memcached/blob/1.4.2/doc/protocol.txt

class swift.common.memcached.**MemcacheConnPool**(*server*, *size*, *connect_timeout*, *tls_context=None*)

Bases: eventlet.pools.Pool

Connection pool for Memcache Connections

The *server* parameter can be a hostname, an IPv4 address, or an IPv6 address with an optional port. See *swift.common.utils.parse_socket_string()* for details.

create()

Generate a new pool item. In order for the pool to function, either this method must be overriden in a subclass or the pool must be constructed with the *create* argument. It accepts no arguments and returns a single instance of whatever thing the pool is supposed to contain.

In general, *create()* is called whenever the pool exceeds its previous high-water mark of concurrently-checked-out-items. In other words, in a new pool with *min_size* of 0, the very first call to *get()* will result in a call to *create()*. If the first caller calls put() before some other caller calls *get()*, then the first item will be returned, and *create()* will not be called a second time.

get()

Return an item from the pool, when one is available. This may cause the calling greenthread to block.

exception swift.common.memcached.**MemcacheConnectionError**

Bases: Exception

exception swift.common.memcached.**MemcachePoolTimeout**(*seconds=None*, *exception=None*)

Bases: eventlet.timeout.Timeout

class swift.common.memcached.**MemcacheRing**(*servers*, *connect_timeout=0.3*, *io_timeout=2.0*, *pool_timeout=1.0*, *tries=3*, *max_conns=2*, *tls_context=None*, *logger=None*, *error_limit_count=10*, *error_limit_time=60*, *error_limit_duration=60*, *item_size_warning_threshold=-1*)

Bases: object

Simple, consistent-hashed memcache client.

decr(*key*, *delta=1*, *time=0*)

Decrements a key which has a numeric value by delta. Calls incr with -delta.

Parameters

- **key** key
- **delta** amount to subtract to the value of key (or set the value to 0 if the key is not found) will be cast to an int
- **time** the time to live

Returns result of decrementing

Raises *MemcacheConnectionError*

delete(*key*, *server_key=None*)

Deletes a key/value pair from memcache.

Parameters

- **key** key to be deleted
- **server_key** key to use in determining which server in the ring is used

get(*key*, *raise_on_error=False*)

Gets the object specified by key. It will also unserialize the object before returning if it is serialized in memcache with JSON.

Parameters

- **key** key
- **raise_on_error** if True, propagate Timeouts and other errors. By default, errors are treated as cache misses.

Returns value of the key in memcache

get_multi(*keys*, *server_key*)

Gets multiple values from memcache for the given keys.

Parameters

- **keys** keys for values to be retrieved from memcache
- **server_key** key to use in determining which server in the ring is used

Returns list of values

incr(*key*, *delta=1*, *time=0*)

Increments a key which has a numeric value by delta. If the key cant be found, its added as delta or 0 if delta < 0. If passed a negative number, will use memcacheds decr. Returns the int stored in memcached Note: The data memcached stores as the result of incr/decr is an unsigned int. decrs that result in a number below 0 are stored as 0.

Parameters

- **key** key
- **delta** amount to add to the value of key (or set as the value if the key is not found) will be cast to an int
- **time** the time to live

Returns result of incrementing

Raises *MemcacheConnectionError*

set(*key*, *value*, *serialize=True*, *time=0*, *min_compress_len=0*, *raise_on_error=False*)

Set a key/value pair in memcache

Parameters

- **key** key
- **value** value
- **serialize** if True, value is serialized with JSON before sending to memcache
- **time** the time to live
- **min_compress_len** minimum compress length, this parameter was added to keep the signature compatible with python-memcached interface. This implementation ignores it.
- **raise_on_error** if True, propagate Timeouts and other errors. By default, errors are ignored.

set_multi(*mapping*, *server_key*, *serialize=True*, *time=0*, *min_compress_len=0*)

Sets multiple key/value pairs in memcache.

Parameters

- **mapping** dictionary of keys and values to be set in memcache
- **server_key** key to use in determining which server in the ring is used
- **serialize** if True, value is serialized with JSON before sending to memcache.

- **time** the time to live

Min_compress_len minimum compress length, this parameter was added to keep the signature compatible with python-memcached interface. This implementation ignores it

swift.common.memcached.**sanitize_timeout**(*timeout*)

Sanitize a timeout value to use an absolute expiration time if the delta is greater than 30 days (in seconds). Note that the memcached server translates negative values to mean a delta of 30 days in seconds (and 1 additional second), client beware.

9.7.11 Middleware Registry

swift.common.registry.**get_sensitive_headers**()

Returns the set of registered sensitive headers.

Used by *swift.common.middleware.proxy_logging* to perform redactions prior to logging.

swift.common.registry.**get_sensitive_params**()

Returns the set of registered sensitive query parameters.

Used by *swift.common.middleware.proxy_logging* to perform redactions prior to logging.

swift.common.registry.**get_swift_info**(*admin=False*, *disallowed_sections=None*)

Returns information about the swift cluster that has been previously registered with the register_swift_info call.

Parameters

- **admin** boolean value, if True will additionally return an admin section with information previously registered as admin info.
- **disallowed_sections** list of section names to be withheld from the information returned.

Returns dictionary of information about the swift cluster.

swift.common.registry.**register_sensitive_header**(*header*)

Register a header as being sensitive.

Sensitive headers are automatically redacted when logging. See the `reveal_sensitive_prefix` option in the proxy-server sample config for more information.

Parameters **header** The (case-insensitive) header name which, if present, may contain sensitive information. Examples include `X-Auth-Token` and (if s3api is enabled) `Authorization`. Limited to ASCII characters.

swift.common.registry.**register_sensitive_param**(*query_param*)

Register a query parameter as being sensitive.

Sensitive query parameters are automatically redacted when logging. See the `reveal_sensitive_prefix` option in the proxy-server sample config for more information.

Parameters **query_param** The (case-sensitive) query parameter name which, if present, may contain sensitive information. Examples include `temp_url_signature` and (if s3api is enabled) `X-Amz-Signature`. Limited to ASCII characters.

swift.common.registry.**register_swift_info**(*name='swift'*, *admin=False*, ***kwargs*)

Registers information about the swift cluster to be retrieved with calls to get_swift_info.

NOTE: Do not use . in the param: name or any keys in kwargs. . is used in the disallowed_sections to remove unwanted keys from /info.

Parameters

- **name** string, the section name to place the information under.
- **admin** boolean, if True, information will be registered to an admin section which can optionally be withheld when requesting the information.
- **kwargs** key value arguments representing the information to be added.

Raises **ValueError** if name or any of the keys in kwargs has . in it

9.7.12 Request Helpers

Miscellaneous utility functions for use in generating responses.

Why not swift.common.utils, you ask? Because this way we can import things from swob in here without creating circular imports.

class swift.common.request_helpers.**SegmentedIterable**(*req*, *app*, *listing_iter*, *max_get_time*, *logger*, *ua_suffix*, *swift_source*, *name='<not specified>'*, *response_body_length=None*)

Bases: object

Iterable that returns the object contents for a large object.

Parameters

- **req** original request object
- **app** WSGI application from which segments will come
- **listing_iter** iterable yielding the object segments to fetch, along with the byte subranges to fetch, in the form of a 5-tuple (object-path, object-etag, object-size, first-byte, last-byte).

  If object-etag is None, no MD5 verification will be done.

  If object-size is None, no length verification will be done.

  If first-byte and last-byte are None, then the entire object will be fetched.
- **max_get_time** maximum permitted duration of a GET request (seconds)
- **logger** logger object
- **swift_source** value of swift.source in subrequest environ (just for logging)
- **ua_suffix** string to append to user-agent.
- **name** name of manifest (used in logging only)
- **response_body_length** optional response body length for the response being sent to the client.

app_iter_range(**a*, ***kw*)

swob.Response will only respond with a 206 status in certain cases; one of those is if the body iterator responds to .app_iter_range().

However, this object (or really, its listing iter) is smart enough to handle the range stuff internally, so we just no-op this out for swob.

app_iter_ranges(*ranges*, *content_type*, *boundary*, *content_size*)

This method assumes that iter(self) yields all the data bytes that go into the response, but none of the MIME stuff. For example, if the response will contain three MIME docs with data abcd, efgh, and ijkl, then iter(self) will give out the bytes abcdefghijkl.

This method inserts the MIME stuff around the data bytes.

close()

Called when the client disconnect. Ensure that the connection to the backend server is closed.

validate_first_segment()

Start fetching object data to ensure that the first segment (if any) is valid. This is to catch cases like first segment is missing or first segments etag doesnt match manifest.

Note: this does not validate that you have any segments. A zero-segment large object is not erroneous; it is just empty.

swift.common.request_helpers.**check_path_header**(*req*, *name*, *length*, *error_msg*)

Validate that the value of path-like header is well formatted. We assume the caller ensures that specific header is present in req.headers.

Parameters

- **req** HTTP request object
- **name** header name
- **length** length of path segment check
- **error_msg** error message for client

Returns A tuple with path parts according to length

Raise HTTPPreconditionFailed if header value is not well formatted.

swift.common.request_helpers.**constrain_req_limit**(*req*, *constrained_limit*)

swift.common.request_helpers.**copy_header_subset**(*from_r*, *to_r*, *condition*)

Will copy desired subset of headers from from_r to to_r.

Parameters

- **from_r** a swob Request or Response
- **to_r** a swob Request or Response
- **condition** a function that will be passed the header key as a single argument and should return True if the header is to be copied.

swift.common.request_helpers.**get_container_update_override_key**(*key*)

Returns the full X-Object-Sysmeta-Container-Update-Override-* header key.

Parameters **key** the key you want to override in the container update

Returns the full header key

swift.common.request_helpers.**get_ip_port**(*node*, *headers*)

swift.common.request_helpers.**get_name_and_placement**(*request*, *minsegs=1*, *maxsegs=None*, *rest_with_last=False*)

Utility function to split and validate the request path and storage policy. The storage policy index is extracted from the headers of the request and converted to a StoragePolicy instance. The remaining args are passed through to *split_and_validate_path()*.

Returns a list, result of *split_and_validate_path()* with the BaseStoragePolicy instance appended on the end

Raises **HTTPServiceUnavailable** if the path is invalid or no policy exists with the extracted policy_index.

swift.common.request_helpers.**get_object_transient_sysmeta**(*key*)

Returns the Object Transient System Metadata header for key. The Object Transient System Metadata namespace will be persisted by backend object servers. These headers are treated in the same way as object user metadata i.e. all headers in this namespace will be replaced on every POST request.

Parameters **key** metadata key

Returns the entire object transient system metadata header for key

swift.common.request_helpers.**get_param**(*req*, *name*, *default=None*)

Get a parameter from an HTTP request ensuring proper handling UTF-8 encoding.

Parameters

- **req** request object
- **name** parameter name
- **default** result to return if the parameter is not found

Returns HTTP request parameter value, as a native string (in py2, as UTF-8 encoded str, not unicode object)

Raises **HTTPBadRequest** if param not valid UTF-8 byte sequence

swift.common.request_helpers.**get_reserved_name**(**parts*)

Generate a valid reserved name that joins the component parts.

Returns a string

swift.common.request_helpers.**get_sys_meta_prefix**(*server_type*)

Returns the prefix for system metadata headers for given server type.

This prefix defines the namespace for headers that will be persisted by backend servers.

Parameters **server_type** type of backend server i.e. [account|container|object]

Returns prefix string for server types system metadata headers

swift.common.request_helpers.**get_user_meta_prefix**(*server_type*)

Returns the prefix for user metadata headers for given server type.

This prefix defines the namespace for headers that will be persisted by backend servers.

Parameters **server_type** type of backend server i.e. [account|container|object]

Returns prefix string for server types user metadata headers

swift.common.request_helpers.**http_response_to_document_iters**(*response*, *read_chunk_size=4096*)

Takes a successful object-GET HTTP response and turns it into an iterator of (first-byte, last-byte, length, headers, body-file) 5-tuples.

The response must either be a 200 or a 206; if you feed in a 204 or something similar, this probably wont work.

Parameters **response** HTTP response, like from bufferedhttp.http_connect(), not a swob.Response.

swift.common.request_helpers.**is_object_transient_sysmeta**(*key*)

Tests if a header key starts with and is longer than the prefix for object transient system metadata.

Parameters **key** header key

Returns True if the key satisfies the test, False otherwise

swift.common.request_helpers.**is_sys_meta**(*server_type*, *key*)

Tests if a header key starts with and is longer than the system metadata prefix for given server type.

Parameters

- **server_type** type of backend server i.e. [account|container|object]
- **key** header key

Returns True if the key satisfies the test, False otherwise

swift.common.request_helpers.**is_sys_or_user_meta**(*server_type*, *key*)

Tests if a header key starts with and is longer than the user or system metadata prefix for given server type.

Parameters

- **server_type** type of backend server i.e. [account|container|object]
- **key** header key

Returns True if the key satisfies the test, False otherwise

swift.common.request_helpers.**is_user_meta**(*server_type*, *key*)

Tests if a header key starts with and is longer than the user metadata prefix for given server type.

Parameters

- **server_type** type of backend server i.e. [account|container|object]
- **key** header key

Returns True if the key satisfies the test, False otherwise

swift.common.request_helpers.**remove_items**(*headers*, *condition*)

Removes items from a dict whose keys satisfy the given condition.

Parameters

- **headers** a dict of headers

- **condition** a function that will be passed the header key as a single argument and should return True if the header is to be removed.

Returns a dict, possibly empty, of headers that have been removed

swift.common.request_helpers.**resolve_etag_is_at_header**(*req*, *metadata*)

Helper function to resolve an alternative etag value that may be stored in metadata under an alternate name.

The value of the requests X-Backend-Etag-Is-At header (if it exists) is a comma separated list of alternate names in the metadata at which an alternate etag value may be found. This list is processed in order until an alternate etag is found.

The left most value in X-Backend-Etag-Is-At will have been set by the left most middleware, or if no middleware, by ECObjectController, if an EC policy is in use. The left most middleware is assumed to be the authority on what the etag value of the object content is.

The resolver will work from left to right in the list until it finds a value that is a name in the given metadata. So the left most wins, IF it exists in the metadata.

By way of example, assume the encrypter middleware is installed. If an object is *not* encrypted then the resolver will not find the encrypter middlewares alternate etag sysmeta (X-Object-Sysmeta-Crypto-Etag) but will then find the EC alternate etag (if EC policy). But if the object *is* encrypted then X-Object-Sysmeta-Crypto-Etag is found and used, which is correct because it should be preferred over X-Object-Sysmeta-Ec-Etag.

Parameters

- **req** a swob Request
- **metadata** a dict containing object metadata

Returns an alternate etag value if any is found, otherwise None

swift.common.request_helpers.**split_and_validate_path**(*request*, *minsegs=1*, *maxsegs=None*, *rest_with_last=False*)

Utility function to split and validate the request path.

Returns result of *split_path()* if everythings okay, as native strings

Raises **HTTPBadRequest** if somethings not okay

swift.common.request_helpers.**split_reserved_name**(*name*)

Separate a valid reserved name into the component parts.

Returns a list of strings

swift.common.request_helpers.**strip_object_transient_sysmeta_prefix**(*key*)

Removes the object transient system metadata prefix from the start of a header key.

Parameters **key** header key

Returns stripped header key

swift.common.request_helpers.**strip_sys_meta_prefix**(*server_type*, *key*)

Removes the system metadata prefix for a given server type from the start of a header key.

Parameters

- **server_type** type of backend server i.e. [account|container|object]

- **key** header key

Returns stripped header key

swift.common.request_helpers.**strip_user_meta_prefix**(*server_type*, *key*)

Removes the user metadata prefix for a given server type from the start of a header key.

Parameters

- **server_type** type of backend server i.e. [account|container|object]
- **key** header key

Returns stripped header key

swift.common.request_helpers.**update_etag_is_at_header**(*req*, *name*)

Helper function to update an X-Backend-Etag-Is-At header whose value is a list of alternative header names at which the actual object etag may be found. This informs the object server where to look for the actual object etag when processing conditional requests.

Since the proxy server and/or middleware may set alternative etag header names, the value of X-Backend-Etag-Is-At is a comma separated list which the object server inspects in order until it finds an etag value.

Parameters

- **req** a swob Request
- **name** name of a sysmeta where alternative etag may be found

swift.common.request_helpers.**update_ignore_range_header**(*req*, *name*)

Helper function to update an X-Backend-Ignore-Range-If-Metadata-Present header whose value is a list of header names which, if any are present on an object, mean the object server should respond with a 200 instead of a 206 or 416.

Parameters

- **req** a swob Request
- **name** name of a header which, if found, indicates the proxy will want the whole object

swift.common.request_helpers.**validate_container_params**(*req*)

swift.common.request_helpers.**validate_internal_account**(*account*)

Validate internal account name.

Raises HTTPBadRequest

swift.common.request_helpers.**validate_internal_container**(*account*, *container*)

Validate internal account and container names.

Raises HTTPBadRequest

swift.common.request_helpers.**validate_internal_obj**(*account*, *container*, *obj*)

Validate internal account, container and object names.

Raises HTTPBadRequest

swift.common.request_helpers.**validate_params**(*req*, *names*)

Get list of parameters from an HTTP request, validating the encoding of each parameter.

Parameters

- **req** request object
- **names** parameter names

Returns a dict mapping parameter names to values for each name that appears in the request parameters

Raises **HTTPBadRequest** if any parameter value is not a valid UTF-8 byte sequence

9.7.13 Swob

Implementation of WSGI Request and Response objects.

This library has a very similar API to Webob. It wraps WSGI request environments and response values into objects that are more friendly to interact with.

Why Swob and not just use WebOb? By Michael Barton

We used webob for years. The main problem was that the interface wasnt stable. For a while, each of our several test suites required a slightly different version of webob to run, and none of them worked with the then-current version. It was a huge headache, so we just scrapped it.

This is kind of a ton of code, but its also been a huge relief to not have to scramble to add a bunch of code branches all over the place to keep Swift working every time webob decides some interface needs to change.

class swift.common.swob.**Accept**(*headerval*)

Bases: object

Wraps a Requests Accept header as a friendly object.

Parameters **headerval** value of the header as a str

best_match(*options*)

Returns the item from options that best matches the accept header. Returns None if no available options are acceptable to the client.

Parameters **options** a list of content-types the server can respond with

Raises **ValueError** if the header is malformed

exception swift.common.swob.**HTTPException**(**args*, ***kwargs*)

Bases: *swift.common.swob.Response*, Exception

class swift.common.swob.**HeaderEnvironProxy**(*environ*)

Bases: collections.abc.MutableMapping

A dict-like object that proxies requests to a wsgi environ, rewriting header keys to environ keys.

For example, headers[Content-Range] sets and gets the value of headers.environ[HTTP_CONTENT_RANGE]

keys() → a set-like object providing a view on D's keys

class swift.common.swob.**Match**(*headerval*)

Bases: object

Wraps a Requests If-[None-]Match header as a friendly object.

Parameters **headerval** value of the header as a str

class swift.common.swob.**Range**(*headerval*)

Bases: object

Wraps a Requests Range header as a friendly object. After initialization, range.ranges is populated with a list of (start, end) tuples denoting the requested ranges.

If there were any syntactically-invalid byte-range-spec values, the constructor will raise a ValueError, per the relevant RFC:

The recipient of a byte-range-set that includes one or more syntactically invalid byte-range-spec values MUST ignore the header field that includes that byte-range-set.

According to the RFC 2616 specification, the following cases will be all considered as syntactically invalid, thus, a ValueError is thrown so that the range header will be ignored. If the range value contains at least one of the following cases, the entire range is considered invalid, ValueError will be thrown so that the header will be ignored.

1. value not starts with bytes=
2. range value start is greater than the end, eg. bytes=5-3
3. range does not have start or end, eg. bytes=-
4. range does not have hyphen, eg. bytes=45
5. range value is non numeric
6. any combination of the above

Every syntactically valid range will be added into the ranges list even when some of the ranges may not be satisfied by underlying content.

Parameters **headerval** value of the header as a str

ranges_for_length(*length*)

This method is used to return multiple ranges for a given length which should represent the length of the underlying content. The constructor method __init__ made sure that any range in ranges list is syntactically valid. So if length is None or size of the ranges is zero, then the Range header should be ignored which will eventually make the response to be 200.

If an empty list is returned by this method, it indicates that there are unsatisfiable ranges found in the Range header, 416 will be returned.

if a returned list has at least one element, the list indicates that there is at least one range valid and the server should serve the request with a 206 status code.

The start value of each range represents the starting position in the content, the end value represents the ending position. This method purposely adds 1 to the end number because the spec defines the Range to be inclusive.

The Range spec can be found at the following link: http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.35.1

Parameters **length** length of the underlying content

class swift.common.swob.**Request**(*environ*)

Bases: object

WSGI Request object.

property accept

Retrieve and set the accept property in the WSGI environ, as a Accept object

property acl

Get and set the swob.ACL property in the WSGI environment

classmethod blank(*path*, *environ=None*, *headers=None*, *body=None*, ***kwargs*)

Create a new request object with the given parameters, and an environment otherwise filled in with non-surprising default values.

Parameters

- **path** encoded, parsed, and unquoted into PATH_INFO
- **environ** WSGI environ dictionary
- **headers** HTTP headers
- **body** stuffed in a WsgiBytesIO and hung on wsgi.input
- **kwargs** any environ key with an property setter

property body

Get and set the request body str

property body_file

Get and set the wsgi.input property in the WSGI environment

call_application(*application*)

Calls the application with this requests environment. Returns the status, headers, and app_iter for the response as a tuple.

Parameters **application** the WSGI application to call

property content_length

Retrieve and set the content-length header as an int

copy_get()

Makes a copy of the request, converting it to a GET.

ensure_x_timestamp()

Similar to *timestamp*, but the X-Timestamp header will be set if not present.

Raises **HTTPBadRequest** if X-Timestamp is already set but not a valid *Timestamp*

Returns the requests X-Timestamp header, as a *Timestamp*

get_response(*application*)

Calls the application with this requests environment. Returns a Response object that wraps up the applications result.

Parameters **application** the WSGI application to call

property host

Get and set the HTTP_HOST property in the WSGI environment

property host_url

Get url for request/response up to path

property if_match

Retrieve and set the if-match property in the WSGI environ, as a Match object

property if_modified_since

Retrieve and set the if-modified-since header as a datetime, set it with a datetime, int, or str

property if_none_match

Retrieve and set the if-none-match property in the WSGI environ, as a Match object

property if_unmodified_since

Retrieve and set the if-unmodified-since header as a datetime, set it with a datetime, int, or str

message_length()

Properly determine the message length for this request. It will return an integer if the headers explicitly contain the message length, or None if the headers dont contain a length. The ValueError exception will be raised if the headers are invalid.

Raises

- **ValueError** if either transfer-encoding or content-length headers have bad values
- **AttributeError** if the last value of the transfer-encoding header is not chunked

property method

Get and set the REQUEST_METHOD property in the WSGI environment

property params

Provides QUERY_STRING parameters as a dictionary

property path

Provides the full path of the request, excluding the QUERY_STRING

property path_info

Get and set the PATH_INFO property in the WSGI environment

path_info_pop()

Takes one path portion (delineated by slashes) from the path_info, and appends it to the script_name. Returns the path segment.

property path_qs

The path of the request, without host but with query string.

property query_string

Get and set the QUERY_STRING property in the WSGI environment

property range

Retrieve and set the range property in the WSGI environ, as a Range object

property referer

Get and set the HTTP_REFERER property in the WSGI environment

property referrer

Get and set the HTTP_REFERER property in the WSGI environment

property remote_addr

Get and set the REMOTE_ADDR property in the WSGI environment

property remote_user

Get and set the REMOTE_USER property in the WSGI environment

property script_name

Get and set the SCRIPT_NAME property in the WSGI environment

split_path(*minsegs=1*, *maxsegs=None*, *rest_with_last=False*)

Validate and split the Requests path.

Examples:

```
['a'] = split_path('/a')
['a', None] = split_path('/a', 1, 2)
['a', 'c'] = split_path('/a/c', 1, 2)
['a', 'c', 'o/r'] = split_path('/a/c/o/r', 1, 3, True)
```

Parameters

- **minsegs** Minimum number of segments to be extracted
- **maxsegs** Maximum number of segments to be extracted
- **rest_with_last** If True, trailing data will be returned as part of last segment. If False, and there is trailing data, raises ValueError.

Returns list of segments with a length of maxsegs (non-existent segments will return as None)

Raises **ValueError** if given an invalid path

property str_params

Provides QUERY_STRING parameters as a dictionary

property swift_entity_path

Provides the (native string) account/container/object path, sans API version.

This can be useful when constructing a path to send to a backend server, as that path will need everything after the /v1.

property timestamp

Provides HTTP_X_TIMESTAMP as a *Timestamp*

property url

Provides the full url of the request

property user_agent

Get and set the HTTP_USER_AGENT property in the WSGI environment

class swift.common.swob.**Response**(*body=None*, *status=200*, *headers=None*, *app_iter=None*, *request=None*, *conditional_response=False*, *conditional_etag=None*, ***kw*)

Bases: object

WSGI Response object.

__call__(*env*, *start_response*)

Respond to the WSGI request.

> **Warning:** This will translate any relative Location header value to an absolute URL using the WSGI environments HOST_URL as a prefix, as RFC 2616 specifies.
>
> However, it is quite common to use relative redirects, especially when it is difficult to know the exact HOST_URL the browser would have used when behind several CNAMEs, CDN services, etc. All modern browsers support relative redirects.
>
> To skip over RFC enforcement of the Location header value, you may set `env['swift.leave_relative_location'] = True` in the WSGI environment.

absolute_location()

Attempt to construct an absolute location.

property accept_ranges

Retrieve and set the accept-ranges header

property app_iter

Retrieve and set the response app_iter

property body

Retrieve and set the Response body str

property charset

Retrieve and set the response charset

property conditional_etag

The conditional_etag keyword argument for Response will allow the conditional match value of a If-Match request to be compared to a non-standard value.

This is available for Storage Policies that do not store the client object data verbatim on the storage nodes, but still need support conditional requests.

Its most effectively used with X-Backend-Etag-Is-At which would define the additional Metadata key(s) where the original ETag of the clear-form client request data may be found.

property content_length

Retrieve and set the content-length header as an int

property content_range

Retrieve and set the content-range header

property content_type

Retrieve and set the response Content-Type header

property etag

Retrieve and set the response Etag header

fix_conditional_response()

You may call this once you have set the content_length to the whole object length and body or app_iter to reset the content_length properties on the request.

It is ok to not call this method, the conditional response will be maintained for you when you __call__ the response.

property host_url

Get url for request/response up to path

property last_modified

Retrieve and set the last-modified header as a datetime, set it with a datetime, int, or str

property location

Retrieve and set the location header

property status

Retrieve and set the Response status, e.g. 200 OK

www_authenticate()

Construct a suitable value for WWW-Authenticate response header

If we have a request and a valid-looking path, the realm is the account; otherwise we set it to unknown.

class swift.common.swob.**StatusMap**

Bases: object

A dict-like object that returns HTTPException subclasses/factory functions where the given key is the status code.

class swift.common.swob.**WsgiBytesIO**(*initial_bytes=b''*)

Bases: _io.BytesIO

This class adds support for the additional wsgi.input methods defined on eventlet.wsgi.Input to the BytesIO class which would otherwise be a fine stand-in for the file-like object in the WSGI environment.

swift.common.swob.**wsgify**(*func*)

A decorator for translating functions which take a swob Request object and return a Response object into WSGI callables. Also catches any raised HTTPExceptions and treats them as a returned Response.

9.7.14 Utils

Miscellaneous utility functions for use with Swift.

swift.common.utils.**ATTRIBUTES_RE = re.compile('(\\w+)=(".*?"|[^";]+)(; ?|$)')**

Regular expression to match form attributes.

class swift.common.utils.**CloseableChain**(**iterables*)

Bases: object

Like itertools.chain, but with a close method that will attempt to invoke its sub-iterators close methods, if any.

class swift.common.utils.**ContextPool**(*size=1000*)

Bases: eventlet.greenpool.GreenPool

GreenPool subclassed to kill its coros when it gets gced

class swift.common.utils.**Everything**

Bases: object

A container that contains everything. If e is an instance of Everything, then x in e is true for all x.

class swift.common.utils.**GreenAsyncPile**(*size_or_pool*)

Bases: object

Runs jobs in a pool of green threads, and the results can be retrieved by using this object as an iterator.

This is very similar in principle to eventlet.GreenPile, except it returns results as they become available rather than in the order they were launched.

Correlating results with jobs (if necessary) is left to the caller.

spawn(*func*, **args*, ***kwargs*)

Spawn a job in a green thread on the pile.

waitall(*timeout*)

Wait timeout seconds for any results to come in.

Parameters **timeout** seconds to wait for results

Returns list of results accrued in that time

waitfirst(*timeout*)

Wait up to timeout seconds for first result to come in.

Parameters **timeout** seconds to wait for results

Returns first item to come back, or None

exception swift.common.utils.**GreenAsyncPileWaitallTimeout**(*seconds=None*, *exception=None*)

Bases: eventlet.timeout.Timeout

class swift.common.utils.**GreenthreadSafeIterator**(*unsafe_iterable*)

Bases: object

Wrap an iterator to ensure that only one greenthread is inside its next() method at a time.

This is useful if an iterators next() method may perform network IO, as that may trigger a greenthread context switch (aka trampoline), which can give another greenthread a chance to call next(). At that point, you get an error like ValueError: generator already executing. By wrapping calls to next() with a mutex, we avoid that error.

class swift.common.utils.**InputProxy**(*wsgi_input*)

Bases: object

File-like object that counts bytes read. To be swapped in for wsgi.input for accounting purposes.

read(**args*, ***kwargs*)

Pass read request to the underlying file-like object and add bytes read to total.

readline(**args*, ***kwargs*)

Pass readline request to the underlying file-like object and add bytes read to total.

exception swift.common.utils.**InvalidHashPathConfigError**

Bases: ValueError

class swift.common.utils.**LRUCache**(*maxsize=1000*, *maxtime=3600*)

Bases: object

Decorator for size/time bound memoization that evicts the least recently used members.

class swift.common.utils.**LogAdapter**(*logger*, *server*)

Bases: logging.LoggerAdapter, object

A Logger like object which performs some reformatting on calls to *exception()*. Can be used to store a threadlocal transaction id and client ip.

exception(*msg*, **args*, ***kwargs*)

Delegate an exception call to the underlying logger.

getEffectiveLevel()

Get the effective level for the underlying logger.

notice(*msg*, **args*, ***kwargs*)

Convenience function for syslog priority LOG_NOTICE. The python logging lvl is set to 25, just above info. SysLogHandler is monkey patched to map this log lvl to the LOG_NOTICE syslog priority.

process(*msg*, *kwargs*)

Add extra info to message

set_statsd_prefix(*prefix*)

This method is deprecated. Callers should use the `statsd_tail_prefix` argument of `get_logger` when instantiating a logger.

The StatsD client prefix defaults to the name of the logger. This method may override that default with a specific value. Currently used in the proxy-server to differentiate the Account, Container, and Object controllers.

statsd_delegate()

Factory to create methods which delegate to methods on self.logger.statsd_client (an instance of StatsdClient). The created methods conditionally delegate to a method whose name is given in statsd_func_name. The created delegate methods are a no-op when StatsD logging is not configured.

Parameters **statsd_func_name** the name of a method on StatsdClient.

class swift.common.utils.**LogLevelFilter**(*level=10*)

Bases: object

Drop messages for the logger based on level.

This is useful when dependencies log too much information.

Parameters **level** All messages at or below this level are dropped (DEBUG < INFO < WARN < ERROR < CRITICAL|FATAL) Default: DEBUG

class swift.common.utils.**MetricsPrefixLoggerAdapter**(*logger*, *extra*, *metric_prefix*)

Bases: *swift.common.utils.SwiftLoggerAdapter*

Adds a prefix to all Statsd metrics names.

swift.common.utils.**NR_ioprio_set**()

Give __NR_ioprio_set value for your system.

class swift.common.utils.**NicerInterpolation**

Bases: configparser.BasicInterpolation

class swift.common.utils.**NullLogger**

Bases: object

A no-op logger for eventlet wsgi.

class swift.common.utils.**OverrideOptions**(*devices*, *partitions*, *policies*)

Bases: tuple

devices

Alias for field number 0

partitions

Alias for field number 1

policies

Alias for field number 2

class swift.common.utils.**PipeMutex**

Bases: object

Mutex using a pipe. Works across both greenlets and real threads, even at the same time.

acquire(*blocking=True*)

Acquire the mutex.

If called with blocking=False, returns True if the mutex was acquired and False if it wasnt. Otherwise, blocks until the mutex is acquired and returns True.

This lock is recursive; the same greenthread may acquire it as many times as it wants to, though it must then release it that many times too.

close()

Close the mutex. This releases its file descriptors.

You cant use a mutex after its been closed.

release()

Release the mutex.

class swift.common.utils.**PrefixLoggerAdapter**(*logger*, *extra*)

Bases: *swift.common.utils.SwiftLoggerAdapter*

Adds an optional prefix to all its log messages. When the prefix has not been set, messages are unchanged.

exception(*msg*, **a*, ***kw*)

Delegate an exception call to the underlying logger.

process(*msg*, *kwargs*)

Process the logging message and keyword arguments passed in to a logging call to insert contextual information. You can either manipulate the message itself, the keyword args or both. Return the message and kwargs modified (or not) to suit your needs.

Normally, youll only need to override this one method in a LoggerAdapter subclass for your specific needs.

class swift.common.utils.**RateLimitedIterator**(*iterable*, *elements_per_second*, *limit_after=0*, *ratelimit_if=<function RateLimitedIterator.<lambda>>*)

Bases: object

Wrap an iterator to only yield elements at a rate of N per second.

Parameters

- **iterable** iterable to wrap
- **elements_per_second** the rate at which to yield elements
- **limit_after** rate limiting kicks in only after yielding this many elements; default is 0 (rate limit immediately)

class swift.common.utils.**ShardRange**(*name*, *timestamp*, *lower=MinBound*, *upper=MaxBound*, *object_count=0*, *bytes_used=0*, *meta_timestamp=None*, *deleted=False*, *state=None*, *state_timestamp=None*, *epoch=None*, *reported=False*, *tombstones=- 1*)

Bases: object

A ShardRange encapsulates sharding state related to a container including lower and upper bounds that define the object namespace for which the container is responsible.

Shard ranges may be persisted in a container database. Timestamps associated with subsets of the shard range attributes are used to resolve conflicts when a shard range needs to be merged with an existing shard range record and the most recent version of an attribute should be persisted.

Parameters

- **name** the name of the shard range; this should take the form of a path to a container i.e. <account_name>/<container_name>.
- **timestamp** a timestamp that represents the time at which the shard ranges lower, upper or deleted attributes were last modified.
- **lower** the lower bound of object names contained in the shard range; the lower bound *is not* included in the shard range namespace.

- **upper** the upper bound of object names contained in the shard range; the upper bound *is* included in the shard range namespace.
- **object_count** the number of objects in the shard range; defaults to zero.
- **bytes_used** the number of bytes in the shard range; defaults to zero.
- **meta_timestamp** a timestamp that represents the time at which the shard ranges `object_count` and `bytes_used` were last updated; defaults to the value of `timestamp`.
- **deleted** a boolean; if True the shard range is considered to be deleted.
- **state** the state; must be one of ShardRange.STATES; defaults to CREATED.
- **state_timestamp** a timestamp that represents the time at which `state` was forced to its current value; defaults to the value of `timestamp`. This timestamp is typically not updated with every change of `state` because in general conflicts in `state` attributes are resolved by choosing the larger `state` value. However, when this rule does not apply, for example when changing state from `SHARDED` to `ACTIVE`, the `state_timestamp` may be advanced so that the new `state` value is preferred over any older `state` value.
- **epoch** optional epoch timestamp which represents the time at which sharding was enabled for a container.
- **reported** optional indicator that this shard and its stats have been reported to the root container.
- **tombstones** the number of tombstones in the shard range; defaults to -1 to indicate that the value is unknown.

class MaxBound

Bases: *swift.common.utils.ShardRangeOuterBound*

class MinBound

Bases: *swift.common.utils.ShardRangeOuterBound*

copy(*timestamp=None*, ***kwargs*)

Creates a copy of the ShardRange.

Parameters **timestamp** (optional) If given, the returned ShardRange will have all of its timestamps set to this value. Otherwise the returned ShardRange will have the original timestamps.

Returns an instance of *ShardRange*

entire_namespace()

Returns True if the ShardRange includes the entire namespace, False otherwise.

expand(*donors*)

Expands the bounds as necessary to match the minimum and maximum bounds of the given donors.

Parameters **donors** A list of *ShardRange*

Returns True if the bounds have been modified, False otherwise.

classmethod from_dict(*params*)

Return an instance constructed using the given dict of params. This method is deliberately less flexible than the class *__init__()* method and requires all of the *__init__()* args to be given in the dict of params.

Parameters **params** a dict of parameters

Returns an instance of this class

includes(*other*)

Returns True if this namespace includes the whole of the other namespace, False otherwise.

Parameters **other** an instance of *ShardRange*

increment_meta(*object_count*, *bytes_used*)

Increment the object stats metadata by the given values and update the meta_timestamp to the current time.

Parameters

- **object_count** should be an integer
- **bytes_used** should be an integer

Raises **ValueError** if `object_count` or `bytes_used` cannot be cast to an int.

classmethod make_path(*shards_account*, *root_container*, *parent_container*, *timestamp*, *index*)

Returns a path for a shard container that is valid to use as a name when constructing a *ShardRange*.

Parameters

- **shards_account** the hidden internal account to which the shard container belongs.
- **root_container** the name of the root container for the shard.
- **parent_container** the name of the parent container for the shard; for initial first generation shards this should be the same as `root_container`; for shards of shards this should be the name of the sharding shard container.
- **timestamp** an instance of *Timestamp*
- **index** a unique index that will distinguish the path from any other path generated using the same combination of `shards_account`, `root_container`, `parent_container` and `timestamp`.

Returns a string of the form <account_name>/<container_name>

overlaps(*other*)

Returns True if the ShardRange namespace overlaps with the other ShardRanges namespace.

Parameters **other** an instance of *ShardRange*

classmethod resolve_state(*state*)

Given a value that may be either the name or the number of a state return a tuple of (state number, state name).

Parameters **state** Either a string state name or an integer state number.

Returns A tuple (state number, state name)

Raises `ValueError` if `state` is neither a valid state name nor a valid state number.

`property row_count`

Returns the total number of rows in the shard range i.e. the sum of objects and tombstones.

Returns the row count

`set_deleted`(*timestamp=None*)

Mark the shard range deleted and set timestamp to the current time.

Parameters `timestamp` optional timestamp to set; if not given the current time will be set.

Returns True if the deleted attribute or timestamp was changed, False otherwise

`update_meta`(*object_count*, *bytes_used*, *meta_timestamp=None*)

Set the object stats metadata to the given values and update the meta_timestamp to the current time.

Parameters

- `object_count` should be an integer
- `bytes_used` should be an integer
- `meta_timestamp` timestamp for metadata; if not given the current time will be set.

Raises `ValueError` if `object_count` or `bytes_used` cannot be cast to an int, or if meta_timestamp is neither None nor can be cast to a *Timestamp*.

`update_state`(*state*, *state_timestamp=None*)

Set state to the given value and optionally update the state_timestamp to the given time.

Parameters

- `state` new state, should be an integer
- `state_timestamp` timestamp for state; if not given the state_timestamp will not be changed.

Returns True if the state or state_timestamp was changed, False otherwise

`update_tombstones`(*tombstones*, *meta_timestamp=None*)

Set the tombstones metadata to the given values and update the meta_timestamp to the current time.

Parameters

- `tombstones` should be an integer
- `meta_timestamp` timestamp for metadata; if not given the current time will be set.

Raises `ValueError` if `tombstones` cannot be cast to an int, or if meta_timestamp is neither None nor can be cast to a *Timestamp*.

class swift.common.utils.**ShardRangeList**(*initlist=None*)

Bases: collections.UserList

This class provides some convenience functions for working with lists of *ShardRange*.

This class does not enforce ordering or continuity of the list items: callers should ensure that items are added in order as appropriate.

property bytes_used

Returns the total number of bytes in all items in the list.

Returns total bytes used

filter(*includes=None*, *marker=None*, *end_marker=None*)

Filter the list for those shard ranges whose namespace includes the `includes` name or any part of the namespace between `marker` and `end_marker`. If none of `includes`, `marker` or `end_marker` are specified then all shard ranges will be returned.

Parameters

- **includes** a string; if not empty then only the shard range, if any, whose namespace includes this string will be returned, and `marker` and `end_marker` will be ignored.
- **marker** if specified then only shard ranges whose upper bound is greater than this value will be returned.
- **end_marker** if specified then only shard ranges whose lower bound is less than this value will be returned.

Returns A new instance of *ShardRangeList* containing the filtered shard ranges.

find_lower(*condition*)

Finds the first shard range satisfies the given condition and returns its lower bound.

Parameters **condition** A function that must accept a single argument of type *ShardRange* and return True if the shard range satisfies the condition or False otherwise.

Returns The lower bound of the first shard range to satisfy the condition, or the `upper` value of this list if no such shard range is found.

includes(*other*)

Check if another ShardRange namespace is enclosed between the lists `lower` and `upper` properties. Note: the lists `lower` and `upper` properties will only equal the outermost bounds of all items in the list if the list has previously been sorted.

Note: the list does not need to contain an item matching `other` for this method to return True, although if the list has been sorted and does contain an item matching `other` then the method will return True.

Parameters **other** an instance of *ShardRange*

Returns True if others namespace is enclosed, False otherwise.

property lower

Returns the lower bound of the first item in the list. Note: this will only be equal to the lowest bound of all items in the list if the list contents has been sorted.

Returns lower bound of first item in the list, or ShardRange.MIN if the list is empty.

property object_count

Returns the total number of objects of all items in the list.

Returns total object count

property row_count

Returns the total number of rows of all items in the list.

Returns total row count

property upper

Returns the upper bound of the first item in the list. Note: this will only be equal to the uppermost bound of all items in the list if the list has previously been sorted.

Returns upper bound of first item in the list, or ShardRange.MIN if the list is empty.

class swift.common.utils.**ShardRangeOuterBound**

Bases: object

A custom singleton type to be subclassed for the outer bounds of ShardRanges.

class swift.common.utils.**Spliterator**(*source_iterable*)

Bases: object

Takes an iterator yielding sliceable things (e.g. strings or lists) and yields subiterators, each yielding up to the requested number of items from the source.

```
>>> si = Spliterator(["abcde", "fg", "hijkl"])
>>> ''.join(si.take(4))
"abcd"
>>> ''.join(si.take(3))
"efg"
>>> ''.join(si.take(1))
"h"
>>> ''.join(si.take(3))
"ijk"
>>> ''.join(si.take(3))
"l"  # shorter than requested; this can happen with the last iterator
```

class swift.common.utils.**StrAnonymizer**(*data*, *method*, *salt*)

Bases: str

Class that permits to get a string anonymized or simply quoted.

class swift.common.utils.**StrFormatTime**(*ts*)

Bases: object

Class that permits to get formats or parts of a time.

class swift.common.utils.**StreamingPile**(*size*)

Bases: *swift.common.utils.GreenAsyncPile*

Runs jobs in a pool of green threads, spawning more jobs as results are retrieved and worker threads become available.

When used as a context manager, has the same worker-killing properties as *ContextPool*.

asyncstarmap(*func*, *args_iter*)

This is the same as `itertools.starmap()`, except that *func* is executed in a separate green thread for each item, and results wont necessarily have the same order as inputs.

class swift.common.utils.**SwiftLogFormatter**(*fmt=None*, *datefmt=None*, *max_line_length=0*)

Bases: `logging.Formatter`

Custom logging.Formatter will append txn_id to a log message if the record has one and the message does not. Optionally it can shorten overly long log lines.

format(*record*)

Format the specified record as text.

The records attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

class swift.common.utils.**SwiftLoggerAdapter**(*logger*, *extra*)

Bases: `logging.LoggerAdapter`

A logging.LoggerAdapter subclass that also passes through StatsD method calls.

Like logging.LoggerAdapter, you have to subclass this and override the process() method to accomplish anything useful.

exception(*msg*, **a*, ***kw*)

Delegate an exception call to the underlying logger.

class swift.common.utils.**ThreadSafeSysLogHandler**(*address=('localhost', 514)*, *facility=1*, *socktype=None*)

Bases: `logging.handlers.SysLogHandler`

createLock()

Acquire a thread lock for serializing access to the underlying I/O.

class swift.common.utils.**Timestamp**(*timestamp*, *offset=0*, *delta=0*, *check_bounds=True*)

Bases: `object`

Internal Representation of Swift Time.

The normalized form of the X-Timestamp header looks like a float with a fixed width to ensure stable string sorting - normalized timestamps look like 1402464677.04188

To support overwrites of existing data without modifying the original timestamp but still maintain consistency a second internal offset vector is append to the normalized timestamp form which compares and sorts greater than the fixed width float format but less than a newer timestamp. The internalized format of timestamps looks like 1402464677.04188_0000000000000000 - the portion after the underscore is the offset and is a formatted hexadecimal integer.

The internalized form is not exposed to clients in responses from Swift. Normal client operations will not create a timestamp with an offset.

The Timestamp class in common.utils supports internalized and normalized formatting of timestamps and also comparison of timestamp values. When the offset value of a Timestamp is 0 - its

considered insignificant and need not be represented in the string format; to support backwards compatibility during a Swift upgrade the internalized and normalized form of a Timestamp with an insignificant offset are identical. When a timestamp includes an offset it will always be represented in the internalized form, but is still excluded from the normalized form. Timestamps with an equivalent timestamp portion (the float part) will compare and order by their offset. Timestamps with a greater timestamp portion will always compare and order greater than a Timestamp with a lesser timestamp regardless of its offset. String comparison and ordering is guaranteed for the internalized string format, and is backwards compatible for normalized timestamps which do not include an offset.

ceil()

Return the normal part of the timestamp rounded up to the nearest integer number of seconds.

This value should be used whenever the second-precision Last-Modified time of a resource is required.

Returns a float value with second precision.

classmethod from_isoformat(*date_string*)

Parse an isoformat string representation of time to a Timestamp object.

Parameters **date_string** a string formatted as per an Timestamp.isoformat property.

Returns an instance of this class.

property isoformat

Get an isoformat string representation of the normal part of the Timestamp with microsecond precision and no trailing timezone, for example:

```
1970-01-01T00:00:00.000000
```

Returns an isoformat string

class swift.common.utils.**Watchdog**

Bases: object

Implements a watchdog to efficiently manage concurrent timeouts.

Compared to eventlet.timeouts.Timeout, it reduces the number of context switching in eventlet by avoiding to schedule actions (throw an Exception), then unschedule them if the timeouts are cancelled.

1. **at T+0, request timeout(10)** => wathdog greenlet sleeps 10 seconds
2. **at T+1, request timeout(15)**

   => the timeout will expire after the current, no need to wake up the watchdog greenlet
3. **at T+2, request timeout(5)**

   => the timeout will expire before the first timeout, wake up the watchdog greenlet to calculate a new sleep period
4. **at T+7, the 3rd timeout expires**

=> the exception is raised, then the greenlet watchdog sleep(3) to wake up for the 1st timeout expiration

`spawn`()

Start the watchdog greenthread.

`start`(*timeout*, *exc*, *timeout_at=None*)

Schedule a timeout action

Parameters

- **timeout** duration before the timeout expires
- **exc** exception to throw when the timeout expire, must inherit from eventlet.timeouts.Timeout
- **timeout_at** allow to force the expiration timestamp

Returns id of the scheduled timeout, needed to cancel it

`stop`(*key*)

Cancel a scheduled timeout

Parameters **key** timeout id, as returned by start()

`class swift.common.utils.WatchdogTimeout`(*watchdog*, *timeout*, *exc*, *timeout_at=None*)

Bases: `object`

Context manager to schedule a timeout in a Watchdog instance

`swift.common.utils.affinity_key_function`(*affinity_str*)

Turns an affinity config value into a function suitable for passing to sort(). After doing so, the array will be sorted with respect to the given ordering.

For example, if affinity_str is r1=1, r2z7=2, r2z8=2, then the array will be sorted with all nodes from region 1 (r1=1) first, then all the nodes from region 2 zones 7 and 8 (r2z7=2 and r2z8=2), then everything else.

Note that the order of the pieces of affinity_str is irrelevant; the priority values are what comes after the equals sign.

If affinity_str is empty or all whitespace, then the resulting function will not alter the ordering of the nodes.

Parameters **affinity_str** affinity config value, e.g. r1z2=3 or r1=1, r2z1=2, r2z2=2

Returns single-argument function

Raises **ValueError** if argument invalid

`swift.common.utils.affinity_locality_predicate`(*write_affinity_str*)

Turns a write-affinity config value into a predicate function for nodes. The returned value will be a 1-arg function that takes a node dictionary and returns a true value if it is local and a false value otherwise. The definition of local comes from the affinity_str argument passed in here.

For example, if affinity_str is r1, r2z2, then only nodes where region=1 or where (region=2 and zone=2) are considered local.

If affinity_str is empty or all whitespace, then the resulting function will consider everything local

Parameters **write_affinity_str** affinity config value, e.g. r1z2 or r1, r2z1, r2z2

Returns single-argument function, or None if affinity_str is empty

Raises **ValueError** if argument invalid

swift.common.utils.**audit_location_generator**(*devices*, *datadir*, *suffix=''*, *mount_check=True*, *logger=None*, *devices_filter=None*, *partitions_filter=None*, *suffixes_filter=None*, *hashes_filter=None*, *hook_pre_device=None*, *hook_post_device=None*, *hook_pre_partition=None*, *hook_post_partition=None*, *hook_pre_suffix=None*, *hook_post_suffix=None*, *hook_pre_hash=None*, *hook_post_hash=None*, *error_counter=None*, *yield_hash_dirs=False*)

Given a devices path and a data directory, yield (path, device, partition) for all files in that directory

(devices|partitions|suffixes|hashes)_filter are meant to modify the list of elements that will be iterated. eg: they can be used to exclude some elements based on a custom condition defined by the caller.

hook_pre_(device|partition|suffix|hash) are called before yielding the element, hook_pos_(device|partition|suffix|hash) are called after the element was yielded. They are meant to do some pre/post processing. eg: saving a progress status.

Parameters

- **devices** parent directory of the devices to be audited
- **datadir** a directory located under self.devices. This should be one of the DATADIR constants defined in the account, container, and object servers.
- **suffix** path name suffix required for all names returned (ignored if yield_hash_dirs is True)
- **mount_check** Flag to check if a mount check should be performed on devices
- **logger** a logger object
- **devices_filter** a callable taking (devices, [list of devices]) as parameters and returning a [list of devices]
- **partitions_filter** a callable taking (datadir_path, [list of parts]) as parameters and returning a [list of parts]
- **suffixes_filter** a callable taking (part_path, [list of suffixes]) as parameters and returning a [list of suffixes]
- **hashes_filter** a callable taking (suff_path, [list of hashes]) as parameters and returning a [list of hashes]
- **hook_pre_device** a callable taking device_path as parameter
- **hook_post_device** a callable taking device_path as parameter

- **hook_pre_partition** a callable taking part_path as parameter
- **hook_post_partition** a callable taking part_path as parameter
- **hook_pre_suffix** a callable taking suff_path as parameter
- **hook_post_suffix** a callable taking suff_path as parameter
- **hook_pre_hash** a callable taking hash_path as parameter
- **hook_post_hash** a callable taking hash_path as parameter
- **error_counter** a dictionary used to accumulate error counts; may add keys unmounted and unlistable_partitions
- **yield_hash_dirs** if True, yield hash dirs instead of individual files

swift.common.utils.**backward**(*f*, *blocksize=4096*)

A generator returning lines from a file starting with the last line, then the second last line, etc. i.e., it reads lines backwards. Stops when the first line (if any) is read. This is useful when searching for recent activity in very large files.

Parameters

- **f** file object to read
- **blocksize** no of characters to go backwards at each block

swift.common.utils.**cache_from_env**(*env*, *allow_none=False*)

Get memcache connection pool from the environment (which had been previously set by the memcache middleware

Parameters **env** wsgi environment dict

Returns swift.common.memcached.MemcacheRing from environment

swift.common.utils.**capture_stdio**(*logger*, ***kwargs*)

Log unhandled exceptions, close stdio, capture stdout and stderr.

param logger: Logger object to use

swift.common.utils.**closing_if_possible**(*maybe_closable*)

Like contextlib.closing(), but doesnt crash if the object lacks a close() method.

PEP 333 (WSGI) says: If the iterable returned by the application has a close() method, the server or gateway must call that method upon completion of the current request[.] This function makes that easier.

swift.common.utils.**compute_eta**(*start_time*, *current_value*, *final_value*)

Compute an ETA. Now only if we could also have a progress bar

Parameters

- **start_time** Unix timestamp when the operation began
- **current_value** Current value
- **final_value** Final value

Returns ETA as a tuple of (length of time, unit of time) where unit of time is one of (h, m, s)

swift.common.utils.**config_auto_int_value**(*value*, *default*)

Returns default if value is None or auto. Returns value as an int or raises ValueError otherwise.

swift.common.utils.**config_fallocate_value**(*reserve_value*)

Returns fallocate reserve_value as an int or float. Returns is_percent as a boolean. Returns a ValueError on invalid fallocate value.

swift.common.utils.**config_positive_int_value**(*value*)

Returns positive int value if it can be cast by int() and its an integer > 0. (not including zero) Raises ValueError otherwise.

swift.common.utils.**config_read_prefixed_options**(*conf*, *prefix_name*, *defaults*)

Read prefixed options from configuration

Parameters

- **conf** the configuration
- **prefix_name** the prefix (including, if needed, an underscore)
- **defaults** a dict of default values. The dict supplies the option name and type (string or comma separated string)

Returns a dict containing the options

swift.common.utils.**config_read_reseller_options**(*conf*, *defaults*)

Read reseller_prefix option and associated options from configuration

Reads the reseller_prefix option, then reads options that may be associated with a specific reseller prefix. Reads options such that an option without a prefix applies to all reseller prefixes unless an option has an explicit prefix.

Parameters

- **conf** the configuration
- **defaults** a dict of default values. The key is the option name. The value is either an array of strings or a string

Returns tuple of an array of reseller prefixes and a dict of option values

swift.common.utils.**config_true_value**(*value*)

Returns True if the value is either True or a string in TRUE_VALUES. Returns False otherwise.

swift.common.utils.**csv_append**(*csv_string*, *item*)

Appends an item to a comma-separated string.

If the comma-separated string is empty/None, just returns item.

swift.common.utils.**decode_timestamps**(*encoded*, *explicit=False*)

Parses a string of the form generated by encode_timestamps and returns a tuple of the three component timestamps. If explicit is False, component timestamps that are not explicitly encoded will be assumed to have zero delta from the previous component and therefore take the value of the previous component. If explicit is True, component timestamps that are not explicitly encoded will be returned with value None.

swift.common.utils.**distribute_evenly**(*items*, *num_buckets*)

Distribute items as evenly as possible into N buckets.

swift.common.utils.**document_iters_to_http_response_body**(*ranges_iter*, *boundary*, *multipart*, *logger*)

Takes an iterator of range iters and turns it into an appropriate HTTP response body, whether thats multipart/byteranges or not.

This is almost, but not quite, the inverse of request_helpers.http_response_to_document_iters(). This function only yields chunks of the body, not any headers.

Parameters

- **ranges_iter** an iterator of dictionaries, one per range. Each dictionary must contain at least the following key: part_iter: iterator yielding the bytes in the range

  Additionally, if multipart is True, then the following other keys are required:

  start_byte: index of the first byte in the range end_byte: index of the last byte in the range content_type: value for the ranges Content-Type header

  Finally, there is one optional key that is used in the multipart/byteranges case:

  entity_length: length of the requested entity (not necessarily equal to the response length). If omitted, * will be used.

  Each part_iter will be exhausted prior to calling next(ranges_iter).

- **boundary** MIME boundary to use, sans dashes (e.g. boundary, not boundary).
- **multipart** True if the response should be multipart/byteranges, False otherwise. This should be True if and only if you have 2 or more ranges.
- **logger** a logger

swift.common.utils.**document_iters_to_multipart_byteranges**(*ranges_iter*, *boundary*)

Takes an iterator of range iters and yields a multipart/byteranges MIME document suitable for sending as the body of a multi-range 206 response.

See document_iters_to_http_response_body for parameter descriptions.

swift.common.utils.**drain_and_close**(*response_or_app_iter*)

Drain and close a swob or WSGI response.

This ensures we dont log a 499 in the proxy just because we realized we dont care about the body of an error.

swift.common.utils.**drop_buffer_cache**(*fd*, *offset*, *length*)

Drop buffer cache for the given range of the given file.

Parameters

- **fd** file descriptor
- **offset** start offset
- **length** length

swift.common.utils.**drop_privileges**(*user*)

Sets the userid/groupid of the current process, get session leader, etc.

Parameters **user** User name to change privileges to

swift.common.utils.**dump_recon_cache**(*cache_dict*, *cache_file*, *logger*, *lock_timeout=2*, *set_owner=None*)

Update recon cache values

Parameters

- **cache_dict** Dictionary of cache key/value pairs to write out
- **cache_file** cache file to update
- **logger** the logger to use to log an encountered error
- **lock_timeout** timeout (in seconds)
- **set_owner** Set owner of recon cache file

swift.common.utils.**encode_timestamps**(*t1*, *t2=None*, *t3=None*, *explicit=False*)

Encode up to three timestamps into a string. Unlike a Timestamp object, the encoded string does NOT used fixed width fields and consequently no relative chronology of the timestamps can be inferred from lexicographic sorting of encoded timestamp strings.

The format of the encoded string is: <t1>[<+/-><t2 - t1>[<+/-><t3 - t2>]]

i.e. if t1 = t2 = t3 then just the string representation of t1 is returned, otherwise the time offsets for t2 and t3 are appended. If explicit is True then the offsets for t2 and t3 are always appended even if zero.

Note: any offset value in t1 will be preserved, but offsets on t2 and t3 are not preserved. In the anticipated use cases for this method (and the inverse decode_timestamps method) the timestamps passed as t2 and t3 are not expected to have offsets as they will be timestamps associated with a POST request. In the case where the encoding is used in a container objects table row, t1 could be the PUT or DELETE time but t2 and t3 represent the content type and metadata times (if different from the data file) i.e. correspond to POST timestamps. In the case where the encoded form is used in a .meta file name, t1 and t2 both correspond to POST timestamps.

swift.common.utils.**eventlet_monkey_patch**()

Install the appropriate Eventlet monkey patches.

swift.common.utils.**expand_ipv6**(*address*)

Expand ipv6 address. :param address: a string indicating valid ipv6 address :returns: a string indicating fully expanded ipv6 address

swift.common.utils.**extract_swift_bytes**(*content_type*)

Parse a content-type and return a tuple containing:

- the content_type string minus any swift_bytes param,
- the swift_bytes value or None if the param was not found

Parameters **content_type** a content-type string

Returns a tuple of (content-type, swift_bytes or None)

swift.common.utils.**fallocate**(*fd*, *size*, *offset=0*)

Pre-allocate disk space for a file.

This function can be disabled by calling disable_fallocate(). If no suitable C function is available in libc, this function is a no-op.

Parameters

- **fd** file descriptor
- **size** size to allocate (in bytes)

swift.common.utils.**fdatasync**(*fd*)

Sync modified file data to disk.

Parameters **fd** file descriptor

swift.common.utils.**filter_shard_ranges**(*shard_ranges*, *includes*, *marker*, *end_marker*)

Filter the given shard ranges to those whose namespace includes the `includes` name or any part of the namespace between `marker` and `end_marker`. If none of `includes`, `marker` or `end_marker` are specified then all shard ranges will be returned.

Parameters

- **shard_ranges** A list of *ShardRange*.
- **includes** a string; if not empty then only the shard range, if any, whose namespace includes this string will be returned, and `marker` and `end_marker` will be ignored.
- **marker** if specified then only shard ranges whose upper bound is greater than this value will be returned.
- **end_marker** if specified then only shard ranges whose lower bound is less than this value will be returned.

Returns A filtered list of *ShardRange*.

swift.common.utils.**find_shard_range**(*item*, *ranges*)

Find a ShardRange in given list of `shard_ranges` whose namespace contains `item`.

Parameters

- **item** The item for a which a ShardRange is to be found.
- **ranges** a sorted list of ShardRanges.

Returns the ShardRange whose namespace contains `item`, or None if no suitable range is found.

swift.common.utils.**fs_has_free_space**(*fs_path*, *space_needed*, *is_percent*)

Check to see whether or not a filesystem has the given amount of space free. Unlike fallocate(), this does not reserve any space.

Parameters

- **fs_path** path to a file or directory on the filesystem; typically the path to the filesystems mount point
- **space_needed** minimum bytes or percentage of free space

- **is_percent** if True, then space_needed is treated as a percentage of the filesystems capacity; if False, space_needed is a number of free bytes.

Returns True if the filesystem has at least that much free space, False otherwise

Raises **OSError** if fs_path does not exist

swift.common.utils.**fsync**(*fd*)

Sync modified file data and metadata to disk.

Parameters **fd** file descriptor

swift.common.utils.**fsync_dir**(*dirpath*)

Sync directory entries to disk.

Parameters **dirpath** Path to the directory to be synced.

swift.common.utils.**get_db_files**(*db_path*)

Given the path to a db file, return a sorted list of all valid db files that actually exist in that paths dir. A valid db filename has the form:

```
<hash>[_<epoch>].db
```

where <hash> matches the <hash> part of the given db_path as would be parsed by `parse_db_filename()`.

Parameters **db_path** Path to a db file that does not necessarily exist.

Returns List of valid db files that do exist in the dir of the `db_path`. This list may be empty.

swift.common.utils.**get_expirer_container**(*x_delete_at*, *expirer_divisor*, *acc*, *cont*, *obj*)

Returns an expiring object container name for given X-Delete-At and (native string) a/c/o.

swift.common.utils.**get_hub**()

Checks whether poll is available and falls back on select if it isnt.

Note about epoll:

Review: https://review.opendev.org/#/c/18806/

There was a problem where once out of every 30 quadrillion connections, a coroutine wouldnt wake up when the client closed its end. Epoll was not reporting the event or it was getting swallowed somewhere. Then when that file descriptor was re-used, eventlet would freak right out because it still thought it was waiting for activity from it in some other coro.

Another note about epoll: its hard to use when forking. epoll works like so:

- create an epoll instance: `efd = epoll_create(...)`
- register file descriptors of interest with `epoll_ctl(efd, EPOLL_CTL_ADD, fd, ...)`
- wait for events with `epoll_wait(efd, ...)`

If you fork, you and all your child processes end up using the same epoll instance, and everyone becomes confused. It is possible to use epoll and fork and still have a correct program as long as you do the right things, but eventlet doesnt do those things. Really, it cant even try to do those things since it doesnt get notified of forks.

In contrast, both poll() and select() specify the set of interesting file descriptors with each call, so theres no problem with forking.

As eventlet monkey patching is now done before call get_hub() in wsgi.py if we use import select we get the eventlet version, but since version 0.20.0 eventlet removed select.poll() function in patched select (see: http://eventlet.net/doc/changelog.html and https://github.com/eventlet/eventlet/commit/614a20462).

We use eventlet.patcher.original function to get python select module to test if poll() is available on platform.

swift.common.utils.**get_log_line**(*req*, *res*, *trans_time*, *additional_info*, *fmt*, *anonymization_method*, *anonymization_salt*)

Make a line for logging that matches the documented log line format for backend servers.

Parameters

- **req** the request.
- **res** the response.
- **trans_time** the time the request took to complete, a float.
- **additional_info** a string to log at the end of the line

Returns a properly formatted line for logging.

swift.common.utils.**get_logger**(*conf*, *name=None*, *log_to_console=False*, *log_route=None*, *fmt='%(server)s: %(message)s'*, *statsd_tail_prefix=None*)

Get the current system logger using config settings.

Log config and defaults:

```
log_facility = LOG_LOCAL0
log_level = INFO
log_name = swift
log_max_line_length = 0
log_udp_host = (disabled)
log_udp_port = logging.handlers.SYSLOG_UDP_PORT
log_address = /dev/log
log_statsd_host = (disabled)
log_statsd_port = 8125
log_statsd_default_sample_rate = 1.0
log_statsd_sample_rate_factor = 1.0
log_statsd_metric_prefix = (empty-string)
```

Parameters

- **conf** Configuration dict to read settings from
- **name** This value is used to populate the `server` field in the log format, as the prefix for statsd messages, and as the default value for `log_route`; defaults to the `log_name` value in `conf`, if it exists, or to swift.
- **log_to_console** Add handler which writes to console on stderr
- **log_route** Route for the logging, not emitted to the log, just used to separate logging configurations; defaults to the value of `name` or whatever `name` defaults to. This value is used as the name attribute of the `logging.LogAdapter` that is returned.

- **fmt** Override log format
- **statsd_tail_prefix** tail prefix to pass to statsd client; if None then the tail prefix defaults to the value of name.

Returns an instance of LogAdapter

swift.common.utils.**get_md5_socket**()

Get an MD5 socket file descriptor. One can MD5 data with it by writing it to the socket with os.write, then os.read the 16 bytes of the checksum out later.

NOTE: It is the callers responsibility to ensure that os.close() is called on the returned file descriptor. This is a bare file descriptor, not a Python object. It doesnt close itself.

swift.common.utils.**get_partition_for_hash**(*hex_hash*, *part_power*)

Return partition number for given hex hash and partition power. :param hex_hash: A hash string :param part_power: partition power :returns: partition number

swift.common.utils.**get_partition_from_path**(*devices*, *path*)

Parameters

- **devices** directory where devices are mounted (e.g. /srv/node)
- **path** full path to a object file or hashdir

Returns the (integer) partition from the path

swift.common.utils.**get_policy_index**(*req_headers*, *res_headers*)

Returns the appropriate index of the storage policy for the request from a proxy server

Parameters

- **req_headers** dict of the request headers.
- **res_headers** dict of the response headers.

Returns string index of storage policy, or None

swift.common.utils.**get_redirect_data**(*response*)

Extract a redirect location from a responses headers.

Parameters **response** a response

Returns a tuple of (path, Timestamp) if a Location header is found, otherwise None

Raises **ValueError** if the Location header is found but a X-Backend-Redirect-Timestamp is not found, or if there is a problem with the format of etiher header

swift.common.utils.**get_time_units**(*time_amount*)

Get a nomralized length of time in the largest unit of time (hours, minutes, or seconds.)

Parameters **time_amount** length of time in seconds

Returns A touple of (length of time, unit of time) where unit of time is one of (h, m, s)

swift.common.utils.**get_valid_utf8_str**(*str_or_unicode*)

Get valid parts of utf-8 str from str, unicode and even invalid utf-8 str

Parameters **str_or_unicode** a string or an unicode which can be invalid utf-8

swift.common.utils.**get_zero_indexed_base_string**(*base*, *index*)

This allows the caller to make a list of things with indexes, where the first item (zero indexed) is just the bare base string, and subsequent indexes are appended -1, -2, etc.

e.g.:

```
'lock', None => 'lock'
'lock', 0    => 'lock'
'lock', 1    => 'lock-1'
'object', 2  => 'object-2'
```

Parameters

- **base** a string, the base string; when `index` is 0 (or None) this is the identity function.
- **index** a digit, typically an integer (or None); for values other than 0 or None this digit is appended to the base string separated by a hyphen.

swift.common.utils.**hash_path**(*account*, *container=None*, *object=None*, *raw_digest=False*)

Get the canonical hash for an account/container/object

Parameters

- **account** Account
- **container** Container
- **object** Object
- **raw_digest** If True, return the raw version rather than a hex digest

Returns hash string

swift.common.utils.**human_readable**(*value*)

Returns the number in a human readable format; for example 1048576 = 1Mi.

swift.common.utils.**is_file_older**(*path*, *age*)

Test if a file mtime is older than the given age, suppressing any OSErrors.

Parameters

- **path** first and only argument passed to os.stat
- **age** age in seconds

Returns True if age is less than or equal to zero or if the file mtime is more than `age` in the past; False if age is greater than zero and the file mtime is less than or equal to `age` in the past or if there is an OSError while stating the file.

swift.common.utils.**is_valid_ip**(*ip*)

Return True if the provided ip is a valid IP-address

swift.common.utils.**is_valid_ipv4**(*ip*)

Return True if the provided ip is a valid IPv4-address

swift.common.utils.**is_valid_ipv6**(*ip*)

Returns True if the provided ip is a valid IPv6-address

swift.common.utils.**ismount**(*path*)

Test whether a path is a mount point. This will catch any exceptions and translate them into a False return value Use ismount_raw to have the exceptions raised instead.

swift.common.utils.**ismount_raw**(*path*)

Test whether a path is a mount point. Whereas ismount will catch any exceptions and just return False, this raw version will not catch exceptions.

This is code hijacked from C Python 2.6.8, adapted to remove the extra lstat() system call.

swift.common.utils.**item_from_env**(*env*, *item_name*, *allow_none=False*)

Get a value from the wsgi environment

Parameters

- **env** wsgi environment dict
- **item_name** name of item to get

Returns the value from the environment

swift.common.utils.**iter_multipart_mime_documents**(*wsgi_input*, *boundary*, *read_chunk_size=4096*)

Given a multi-part-mime-encoded input file object and boundary, yield file-like objects for each part. Note that this does not split each part into headers and body; the caller is responsible for doing that if necessary.

Parameters

- **wsgi_input** The file-like object to read from.
- **boundary** The mime boundary to separate new file-like objects on.

Returns A generator of file-like objects for each part.

Raises *MimeInvalid* if the document is malformed

swift.common.utils.**last_modified_date_to_timestamp**(*last_modified_date_str*)

Convert a last modified date (like youd get from a container listing, e.g. 2014-02-28T23:22:36.698390) to a float.

swift.common.utils.**link_fd_to_path**(*fd*, *target_path*, *dirs_created=0*, *retries=2*, *fsync=True*)

Creates a link to file descriptor at target_path specified. This method does not close the fd for you. Unlike rename, as linkat() cannot overwrite target_path if it exists, we unlink and try again.

Attempts to fix / hide race conditions like empty object directories being removed by backend processes during uploads, by retrying.

Parameters

- **fd** File descriptor to be linked
- **target_path** Path in filesystem where fd is to be linked
- **dirs_created** Number of newly created directories that needs to be fsyncd.
- **retries** number of retries to make
- **fsync** fsync on containing directory of target_path and also all the newly created directories.

swift.common.utils.**list_from_csv**(*comma_separated_str*)

Splits the str given and returns a properly stripped list of the comma separated values.

swift.common.utils.**load_libc_function**(*func_name*, *log_error=True*, *fail_if_missing=False*, *errcheck=False*)

Attempt to find the function in libc, otherwise return a no-op func.

Parameters

- **func_name** name of the function to pull from libc.
- **log_error** log an error when a function cant be found
- **fail_if_missing** raise an exception when a function cant be found. Default behavior is to return a no-op function.
- **errcheck** boolean, if true install a wrapper on the function to check for a return values of -1 and call ctype.get_errno and raise an OSError

swift.common.utils.**load_recon_cache**(*cache_file*)

Load a recon cache file. Treats missing file as empty.

swift.common.utils.**lock_file**(*filename*, *timeout=None*, *append=False*, *unlink=True*)

Context manager that acquires a lock on a file. This will block until the lock can be acquired, or the timeout time has expired (whichever occurs first).

Parameters

- **filename** file to be locked
- **timeout** timeout (in seconds). If None, defaults to DEFAULT_LOCK_TIMEOUT
- **append** True if file should be opened in append mode
- **unlink** True if the file should be unlinked at the end

swift.common.utils.**lock_parent_directory**(*filename*, *timeout=None*)

Context manager that acquires a lock on the parent directory of the given file path. This will block until the lock can be acquired, or the timeout time has expired (whichever occurs first).

Parameters

- **filename** file path of the parent directory to be locked
- **timeout** timeout (in seconds). If None, defaults to DEFAULT_LOCK_TIMEOUT

swift.common.utils.**lock_path**(*directory*, *timeout=None*, *timeout_class=None*, *limit=1*, *name=None*)

Context manager that acquires a lock on a directory. This will block until the lock can be acquired, or the timeout time has expired (whichever occurs first).

For locking exclusively, file or directory has to be opened in Write mode. Python doesnt allow directories to be opened in Write Mode. So we workaround by locking a hidden file in the directory.

Parameters

- **directory** directory to be locked

- **timeout** timeout (in seconds). If None, defaults to DEFAULT_LOCK_TIMEOUT
- **timeout_class** The class of the exception to raise if the lock cannot be granted within the timeout. Will be constructed as timeout_class(timeout, lockpath). Default: LockTimeout
- **limit** The maximum number of locks that may be held concurrently on the same directory at the time this method is called. Note that this limit is only applied during the current call to this method and does not prevent subsequent calls giving a larger limit. Defaults to 1.
- **name** A string to distinguishes different type of locks in a directory

Raises

- **TypeError** if limit is not an int.
- **ValueError** if limit is less than 1.

swift.common.utils.**make_db_file_path**(*db_path*, *epoch*)

Given a path to a db file, return a modified path whose filename part has the given epoch.

A db filename takes the form <hash>[_<epoch>].db; this method replaces the <epoch> part of the given db_path with the given epoch value, or drops the epoch part if the given epoch is None.

Parameters

- **db_path** Path to a db file that does not necessarily exist.
- **epoch** A string (or None) that will be used as the epoch in the new paths filename; non-None values will be normalized to the normal string representation of a *Timestamp*.

Returns A modified path to a db file.

Raises **ValueError** if the epoch is not valid for constructing a *Timestamp*.

swift.common.utils.**makedirs_count**(*path*, *count=0*)

Same as os.makedirs() except that this method returns the number of new directories that had to be created.

Also, this does not raise an error if target directory already exists. This behaviour is similar to Python 3.xs os.makedirs() called with exist_ok=True. Also similar to swift.common.utils.mkdirs()

https://hg.python.org/cpython/file/v3.4.2/Lib/os.py#l212

swift.common.utils.**maybe_multipart_byteranges_to_document_iters**(*app_iter*, *content_type*)

Takes an iterator that may or may not contain a multipart MIME document as well as content type and returns an iterator of body iterators.

Parameters

- **app_iter** iterator that may contain a multipart MIME document
- **content_type** content type of the app_iter, used to determine whether it conains a multipart document and, if so, what the boundary is between documents

swift.common.utils.**md5**(*string=b''*, *usedforsecurity=True*)

Return an md5 hashlib object without usedforsecurity parameter

For python distributions that do not yet support this keyword parameter, we drop the parameter

swift.common.utils.**md5_hash_for_file**(*fname*)

Get the MD5 checksum of a file.

Parameters **fname** path to file

Returns MD5 checksum, hex encoded

swift.common.utils.**mime_to_document_iters**(*input_file*, *boundary*, *read_chunk_size=4096*)

Takes a file-like object containing a multipart MIME document and returns an iterator of (headers, body-file) tuples.

Parameters

- **input_file** file-like object with the MIME doc in it
- **boundary** MIME boundary, sans dashes (e.g. divider, not divider)
- **read_chunk_size** size of strings read via input_file.read()

swift.common.utils.**mkdirs**(*path*)

Ensures the path is a directory or makes it if not. Errors if the path exists but is a file or on permissions failure.

Parameters **path** path to create

swift.common.utils.**modify_priority**(*conf*, *logger*)

Modify priority by nice and ionice.

swift.common.utils.**multipart_byteranges_to_document_iters**(*input_file*, *boundary*, *read_chunk_size=4096*)

Takes a file-like object containing a multipart/byteranges MIME document (see RFC 7233, Appendix A) and returns an iterator of (first-byte, last-byte, length, document-headers, body-file) 5-tuples.

Parameters

- **input_file** file-like object with the MIME doc in it
- **boundary** MIME boundary, sans dashes (e.g. divider, not divider)
- **read_chunk_size** size of strings read via input_file.read()

swift.common.utils.**non_negative_float**(*value*)

Check that the value casts to a float and is non-negative.

Parameters **value** value to check

Raises **ValueError** if the value cannot be cast to a float or is negative.

Returns a float

swift.common.utils.**non_negative_int**(*value*)

Check that the value casts to an int and is a whole number.

Parameters **value** value to check

Raises `ValueError` if the value cannot be cast to an int or does not represent a whole number.

Returns an int

swift.common.utils.**normalize_delete_at_timestamp**(*timestamp*, *high_precision=False*)

Format a timestamp (string or numeric) into a standardized xxxxxxxxxx (10) or xxxxxxxxxx.xxxxx (10.5) format.

Note that timestamps less than 0000000000 are raised to 0000000000 and values greater than November 20th, 2286 at 17:46:39 UTC will be capped at that date and time, resulting in no return value exceeding 9999999999.99999 (or 9999999999 if using low-precision).

This cap is because the expirer is already working through a sorted list of strings that were all a length of 10. Adding another digit would mess up the sort and cause the expirer to break from processing early. By 2286, this problem will need to be fixed, probably by creating an additional .expiring_objects account to work from with 11 (or more) digit container names.

Parameters `timestamp` unix timestamp

Returns normalized timestamp as a string

swift.common.utils.**normalize_timestamp**(*timestamp*)

Format a timestamp (string or numeric) into a standardized xxxxxxxxxx.xxxxx (10.5) format.

Note that timestamps using values greater than or equal to November 20th, 2286 at 17:46 UTC will use 11 digits to represent the number of seconds.

Parameters `timestamp` unix timestamp

Returns normalized timestamp as a string

swift.common.utils.**override_bytes_from_content_type**(*listing_dict*, *logger=None*)

Takes a dict from a container listing and overrides the content_type, bytes fields if swift_bytes is set.

swift.common.utils.**pairs**(*item_list*)

Returns an iterator of all pairs of elements from item_list.

Parameters `item_list` items (no duplicates allowed)

swift.common.utils.**parse_content_disposition**(*header*)

Given the value of a header like: Content-Disposition: form-data; name=somefile; filename=test.html

Return data like (form-data, {name: somefile, filename: test.html})

Parameters `header` Value of a header (the part after the :).

Returns (value name, dict) of the attribute data parsed (see above).

swift.common.utils.**parse_content_range**(*content_range*)

Parse a content-range header into (first_byte, last_byte, total_size).

See RFC 7233 section 4.2 for details on the header format, but its basically Content-Range: bytes ${start}-${end}/${total}.

Parameters `content_range` Content-Range header value to parse, e.g. bytes 100-1249/49004

Returns 3-tuple (start, end, total)

Raises `ValueError` if malformed

`swift.common.utils.`**`parse_content_type`**(*content_type*)

Parse a content-type and its parameters into values. RFC 2616 sec 14.17 and 3.7 are pertinent.

Examples:

```
'text/plain; charset=UTF-8' -> ('text/plain', [('charset, 'UTF-8')])
'text/plain; charset=UTF-8; level=1' ->
    ('text/plain', [('charset, 'UTF-8'), ('level', '1')])
```

Parameters `content_type` content_type to parse

Returns a tuple containing (content type, list of k, v parameter tuples)

`swift.common.utils.`**`parse_db_filename`**(*filename*)

Splits a db filename into three parts: the hash, the epoch, and the extension.

```
>>> parse_db_filename("ab2134.db")
('ab2134', None, '.db')
>>> parse_db_filename("ab2134_1234567890.12345.db")
('ab2134', '1234567890.12345', '.db')
```

Parameters `filename` A db file basename or path to a db file.

Returns A tuple of (hash , epoch, extension). `epoch` may be None.

Raises `ValueError` if `filename` is not a path to a file.

`swift.common.utils.`**`parse_mime_headers`**(*doc_file*)

Takes a file-like object containing a MIME document and returns a HeaderKeyDict containing the headers. The body of the message is not consumed: the position in doc_file is left at the beginning of the body.

This function was inspired by the Python standard librarys http.client.parse_headers.

Parameters `doc_file` binary file-like object containing a MIME document

Returns a swift.common.swob.HeaderKeyDict containing the headers

`swift.common.utils.`**`parse_options`**(*parser=None*, *once=False*, *test_args=None*)

Parse standard swift server/daemon options with optparse.OptionParser.

Parameters

- `parser` OptionParser to use. If not sent one will be created.
- `once` Boolean indicating the once option is available
- `test_args` Override sys.argv; used in testing

Returns Tuple of (config, options); config is an absolute path to the config file, options is the parser options as a dictionary.

Raises `SystemExit` First arg (CONFIG) is required, file must exist

swift.common.utils.**parse_override_options**(***kwargs*)

Figure out which policies, devices, and partitions we should operate on, based on kwargs.

If override_policies is already present in kwargs, then return that value. This happens when using multiple worker processes; the parent process supplies override_policies=X to each child process.

Otherwise, in run-once mode, look at the policies keyword argument. This is the value of the policies command-line option. In run-forever mode or if no policies option was provided, an empty list will be returned.

The procedures for devices and partitions are similar.

Returns a named tuple with fields devices, partitions, and policies.

swift.common.utils.**parse_prefixed_conf**(*conf_file*, *prefix*)

Search the config file for any common-prefix sections and load those sections to a dict mapping the after-prefix reference to options.

Parameters

- **conf_file** the file name of the config to parse
- **prefix** the common prefix of the sections

Returns a dict mapping policy reference -> dict of policy options

Raises **ValueError** if a policy config section has an invalid name

swift.common.utils.**parse_socket_string**(*socket_string*, *default_port*)

Given a string representing a socket, returns a tuple of (host, port). Valid strings are DNS names, IPv4 addresses, or IPv6 addresses, with an optional port. If an IPv6 address is specified it **must** be enclosed in [], like *[::1]* or *[::1]:11211*. This follows the accepted prescription for IPv6 host literals.

Examples:

```
server.org
server.org:1337
127.0.0.1:1337
[::1]:1337
[::1]
```

swift.common.utils.**private**(*func*)

Decorator to declare which methods are privately accessible as HTTP requests with an `X-Backend-Allow-Private-Methods: True` override

Parameters **func** function to make private

swift.common.utils.**public**(*func*)

Decorator to declare which methods are publicly accessible as HTTP requests

Parameters **func** function to make public

swift.common.utils.**punch_hole**(*fd*, *offset*, *length*)

De-allocate disk space in the middle of a file.

Parameters

- **fd** file descriptor

- **offset** index of first byte to de-allocate
- **length** number of bytes to de-allocate

swift.common.utils.**put_recon_cache_entry**(*cache_entry*, *key*, *item*)

Update a recon cache entry item.

If `item` is an empty dict then any existing `key` in `cache_entry` will be deleted. Similarly if `item` is a dict and any of its values are empty dicts then the corrsponsing key will be deleted from the nested dict in `cache_entry`.

We use nested recon cache entries when the object auditor runs in parallel or else in once mode with a specified subset of devices.

Parameters

- **cache_entry** a dict of existing cache entries
- **key** key for item to update
- **item** value for item to update

swift.common.utils.**quorum_size**(*n*)

quorum size as it applies to services that use replication for data integrity (Account/Container services). Object quorum_size is defined on a storage policy basis.

Number of successful backend requests needed for the proxy to consider the client request successful.

swift.common.utils.**quote**(*value*, *safe='/'*)

Patched version of urllib.quote that encodes utf-8 strings before quoting

swift.common.utils.**random**() → x in the interval [0, 1).

swift.common.utils.**ratelimit_sleep**(*running_time*, *max_rate*, *incr_by=1*, *rate_buffer=5*)

Will eventlet.sleep() for the appropriate time so that the max_rate is never exceeded. If max_rate is 0, will not ratelimit. The maximum recommended rate should not exceed (1000 * incr_by) a second as eventlet.sleep() does involve some overhead. Returns running_time that should be used for subsequent calls.

Parameters

- **running_time** the running time in milliseconds of the next allowable request. Best to start at zero.
- **max_rate** The maximum rate per second allowed for the process.
- **incr_by** How much to increment the counter. Useful if you want to ratelimit 1024 bytes/sec and have differing sizes of requests. Must be > 0 to engage rate-limiting behavior.
- **rate_buffer** Number of seconds the rate counter can drop and be allowed to catch up (at a faster than listed rate). A larger number will result in larger spikes in rate but better average accuracy. Must be > 0 to engage rate-limiting behavior.

Returns The absolute time for the next interval in milliseconds; note that time could have passed well beyond that point, but the next call will catch that and skip the sleep.

swift.common.utils.**readconf**(*conf_path*, *section_name=None*, *log_name=None*, *defaults=None*, *raw=False*)

Read config file(s) and return config items as a dict

Parameters

- **conf_path** path to config file/directory, or a file-like object (hasattr readline)
- **section_name** config section to read (will return all sections if not defined)
- **log_name** name to be used with logging (will use section_name if not defined)
- **defaults** dict of default values to pre-populate the config with

Returns dict of config items

Raises

- **ValueError** if section_name does not exist
- **IOError** if reading the file failed

swift.common.utils.**reiterate**(*iterable*)

Consume the first truthy item from an iterator, then re-chain it to the rest of the iterator. This is useful when you want to make sure the prologue to downstream generators have been executed before continuing. :param iterable: an iterable object

swift.common.utils.**remove_directory**(*path*)

Wrapper for os.rmdir, ENOENT and ENOTEMPTY are ignored

Parameters **path** first and only argument passed to os.rmdir

swift.common.utils.**remove_file**(*path*)

Quiet wrapper for os.unlink, OSErrors are suppressed

Parameters **path** first and only argument passed to os.unlink

swift.common.utils.**renamer**(*old*, *new*, *fsync=True*)

Attempt to fix / hide race conditions like empty object directories being removed by backend processes during uploads, by retrying.

The containing directory of new and of all newly created directories are fsyncd by default. This _will_ come at a performance penalty. In cases where these additional fsyncs are not necessary, it is expected that the caller of renamer() turn it off explicitly.

Parameters

- **old** old path to be renamed
- **new** new path to be renamed to
- **fsync** fsync on containing directory of new and also all the newly created directories.

swift.common.utils.**replace_partition_in_path**(*devices*, *path*, *part_power*)

Takes a path and a partition power and returns the same path, but with the correct partition number. Most useful when increasing the partition power.

Parameters

- **devices** directory where devices are mounted (e.g. /srv/node)

- **path** full path to a object file or hashdir
- **part_power** partition power to compute correct partition number

Returns Path with re-computed partition power

swift.common.utils.**replication**(*func*)

Decorator to declare which methods are accessible for different type of servers:

- If option replication_server is None then this decorator doesnt matter.
- If option replication_server is True then ONLY decorated with this decorator methods will be started.
- If option replication_server is False then decorated with this decorator methods will NOT be started.

Parameters **func** function to mark accessible for replication

swift.common.utils.**round_robin_iter**(*its*)

Takes a list of iterators, yield an element from each in a round-robin fashion until all of them are exhausted. :param its: list of iterators

swift.common.utils.**rsync_ip**(*ip*)

Transform ip string to an rsync-compatible form

Will return ipv4 addresses unchanged, but will nest ipv6 addresses inside square brackets.

Parameters **ip** an ip string (ipv4 or ipv6)

Returns a string ip address

swift.common.utils.**rsync_module_interpolation**(*template*, *device*)

Interpolate devices variables inside a rsync module template

Parameters

- **template** rsync module template as a string
- **device** a device from a ring

Returns a string with all variables replaced by device attributes

swift.common.utils.**search_tree**(*root*, *glob_match*, *ext=''*, *exts=None*, *dir_ext=None*)

Look in root, for any files/dirs matching glob, recursively traversing any found directories looking for files ending with ext

Parameters

- **root** start of search path
- **glob_match** glob to match in root, matching dirs are traversed with os.walk
- **ext** only files that end in ext will be returned
- **exts** a list of file extensions; only files that end in one of these extensions will be returned; if set this list overrides any extension specified using the ext param.
- **dir_ext** if present directories that end with dir_ext will not be traversed and instead will be returned as a matched path

Returns list of full paths to matching files, sorted

swift.common.utils.**set_swift_dir**(*swift_dir*)

Sets the directory from which swift config files will be read. If the given directory differs from that already set then the swift.conf file in the new directory will be validated and storage policies will be reloaded from the new swift.conf file.

Parameters **swift_dir** non-default directory to read swift.conf from

class swift.common.utils.**sockaddr_alg**

Bases: _ctypes.Structure

swift.common.utils.**split_path**(*path*, *minsegs=1*, *maxsegs=None*, *rest_with_last=False*)

Validate and split the given HTTP request path.

Examples:

```
['a'] = split_path('/a')
['a', None] = split_path('/a', 1, 2)
['a', 'c'] = split_path('/a/c', 1, 2)
['a', 'c', 'o/r'] = split_path('/a/c/o/r', 1, 3, True)
```

Parameters

- **path** HTTP Request path to be split
- **minsegs** Minimum number of segments to be extracted
- **maxsegs** Maximum number of segments to be extracted
- **rest_with_last** If True, trailing data will be returned as part of last segment. If False, and there is trailing data, raises ValueError.

Returns list of segments with a length of maxsegs (non-existent segments will return as None)

Raises **ValueError** if given an invalid path

swift.common.utils.**storage_directory**(*datadir*, *partition*, *name_hash*)

Get the storage directory

Parameters

- **datadir** Base data directory
- **partition** Partition
- **name_hash** Account, container or object name hash

Returns Storage directory

swift.common.utils.**streq_const_time**(*s1*, *s2*)

Constant-time string comparison.

Params s1 the first string

Params s2 the second string

Returns True if the strings are equal.

This function takes two strings and compares them. It is intended to be used when doing a comparison for authentication purposes to help guard against timing attacks.

swift.common.utils.**strict_b64decode**(*value*, *allow_line_breaks=False*)

Validate and decode Base64-encoded data.

The stdlib base64 module silently discards bad characters, but we often want to treat them as an error.

Parameters

- **value** some base64-encoded data
- **allow_line_breaks** if True, ignore carriage returns and newlines

Returns the decoded data

Raises **ValueError** if value is not a string, contains invalid characters, or has insufficient padding

swift.common.utils.**systemd_notify**(*logger=None*)

Notify the service manager that started this process, if it is systemd-compatible, that this process correctly started. To do so, it communicates through a Unix socket stored in environment variable NOTIFY_SOCKET. More information can be found in systemd documentation: https://www.freedesktop.org/software/systemd/man/sd_notify.html

Parameters **logger** a logger object

swift.common.utils.**timing_stats**(***dec_kwargs*)

Returns a decorator that logs timing events or errors for public methods in swifts wsgi server controllers, based on response code.

swift.common.utils.**unlink_older_than**(*path*, *mtime*)

Remove any file in a given path that was last modified before mtime.

Parameters

- **path** path to remove file from
- **mtime** timestamp of oldest file to keep

swift.common.utils.**unlink_paths_older_than**(*filepaths*, *mtime*)

Remove any files from the given list that were last modified before mtime.

Parameters

- **filepaths** a list of strings, the full paths of files to check
- **mtime** timestamp of oldest file to keep

swift.common.utils.**validate_device_partition**(*device*, *partition*)

Validate that a device and a partition are valid and wont lead to directory traversal when used.

Parameters

- **device** device to validate
- **partition** partition to validate

Raises **ValueError** if given an invalid device or partition

swift.common.utils.**validate_sync_to**(*value*, *allowed_sync_hosts*, *realms_conf*)

Validates an X-Container-Sync-To header value, returning the validated endpoint, realm, and realm_key, or an error string.

Parameters

- **value** The X-Container-Sync-To header value to validate.
- **allowed_sync_hosts** A list of allowed hosts in endpoints, if realms_conf does not apply.
- **realms_conf** An instance of swift.common.container_sync_realms.ContainerSyncRealms to validate against.

Returns A tuple of (error_string, validated_endpoint, realm, realm_key). The error_string will None if the rest of the values have been validated. The validated_endpoint will be the validated endpoint to sync to. The realm and realm_key will be set if validation was done through realms_conf.

swift.common.utils.**whataremyips**(*ring_ip=None*)

Get our IP addresses (us being the set of services configured by one **.conf* file). If our REST listens on a specific address, return it. Otherwise, if listen on 0.0.0.0 or :: return all addresses, including the loopback.

Parameters **ring_ip** (*str*) Optional ring_ip/bind_ip from a config file; may be IP address or hostname.

Returns list of Strings of ip addresses

swift.common.utils.**write_file**(*path*, *contents*)

Write contents to file at path

Parameters

- **path** any path, subdirs will be created as needed
- **contents** data to write to file, will be converted to string

swift.common.utils.**write_pickle**(*obj*, *dest*, *tmp=None*, *pickle_protocol=0*)

Ensure that a pickle file gets written to disk. The file is first written to a tmp location, ensure it is synced to disk, then perform a move to its final location

Parameters

- **obj** python object to be pickled
- **dest** path of final destination file
- **tmp** path to tmp to use, defaults to None
- **pickle_protocol** protocol to pickle the obj with, defaults to 0

9.7.15 WSGI

WSGI tools for use with swift.

class swift.common.wsgi.**ConfigDirLoader**(*conf_dir*)

Bases: *swift.common.wsgi.NamedConfigLoader*

Read configuration from multiple files under the given path.

exception swift.common.wsgi.**ConfigFileError**

Bases: Exception

exception swift.common.wsgi.**ConfigFilePortError**

Bases: *swift.common.wsgi.ConfigFileError*

class swift.common.wsgi.**ConfigString**(*config_string*)

Bases: *swift.common.wsgi.NamedConfigLoader*

Wrap a raw config string up for paste.deploy.

If you give one of these to our loadcontext (e.g. give it to our appconfig) well intercept it and get it routed to the right loader.

class swift.common.wsgi.**NamedConfigLoader**(*filename*)

Bases: paste.deploy.loadwsgi.ConfigLoader

Patch paste.deploys ConfigLoader so each context object will know what config section it came from.

class swift.common.wsgi.**PipelineWrapper**(*context*)

Bases: object

This class provides a number of utility methods for modifying the composition of a wsgi pipeline.

create_filter(*entry_point_name*)

Creates a context for a filter that can subsequently be added to a pipeline context.

Parameters **entry_point_name** entry point of the middleware (Swift only)

Returns a filter context

index(*entry_point_name*)

Returns the first index of the given entry point name in the pipeline.

Raises ValueError if the given module is not in the pipeline.

insert_filter(*ctx*, *index=0*)

Inserts a filter module into the pipeline context.

Parameters

- **ctx** the context to be inserted
- **index** (optional) index at which filter should be inserted in the list of pipeline filters. Default is 0, which means the start of the pipeline.

startswith(*entry_point_name*)

Tests if the pipeline starts with the given entry point name.

Parameters **entry_point_name** entry point of middleware or app (Swift only)

Returns True if entry_point_name is first in pipeline, False otherwise

class swift.common.wsgi.**RestrictedGreenPool**(*size=1024*)

Bases: eventlet.greenpool.GreenPool

Works the same as GreenPool, but if the size is specified as one, then the spawn_n() method will invoke waitall() before returning to prevent the caller from doing any other work (like calling accept()).

spawn_n(**args*, ***kwargs*)

Create a greenthread to run the *function*, the same as spawn(). The difference is that *spawn_n()* returns None; the results of *function* are not retrievable.

class swift.common.wsgi.**ServersPerPortStrategy**(*conf*, *logger*, *servers_per_port*)

Bases: *swift.common.wsgi.StrategyBase*

WSGI server management strategy object for an object-server with one listen port per unique local port in the storage policy rings. The *servers_per_port* integer config setting determines how many workers are run per port.

Tracking data is a map like port -> [(pid, socket), ...].

Used in *run_wsgi()*.

Parameters

- **conf** (*dict*) Server configuration dictionary.
- **logger** The servers LogAdaptor object.
- **servers_per_port** (*int*) The number of workers to run per port.

iter_sockets()

Yields all known listen sockets.

log_sock_exit(*sock*, *data*)

Log a servers exit.

loop_timeout()

Return timeout before checking for reloaded rings.

Returns The time to wait for a child to exit before checking for reloaded rings (new ports).

new_worker_socks()

Yield a sequence of (socket, (port, server_idx)) tuples for each server which should be forked-off and started.

Any sockets for orphaned ports no longer in any ring will be closed (causing their associated workers to gracefully exit) after all new sockets have been yielded.

The server_idx item for each socket will passed into the *log_sock_exit()* and *register_worker_start()* methods.

no_fork_sock()

This strategy does not support running in the foreground.

register_worker_exit(*pid*)

Called when a worker has exited.

Parameters **pid** (*int*) The PID of the worker that exited.

register_worker_start(*sock*, *data*, *pid*)

Called when a new worker is started.

Parameters

- **sock** (*socket*) The listen socket for the worker just started.
- **server_idx** The sockets server_idx as yielded by *new_worker_socks()*.
- **pid** (*int*) The new worker process PID

class swift.common.wsgi.**StrategyBase**(*conf*, *logger*)

Bases: object

Some operations common to all strategy classes.

post_fork_hook()

Called in each forked-off child process, prior to starting the actual wsgi server, to perform any initialization such as drop privileges.

set_close_on_exec_on_listen_sockets()

Set the close-on-exec flag on any listen sockets.

shutdown_sockets()

Shutdown any listen sockets.

signal_ready()

Signal that the server is up and accepting connections.

class swift.common.wsgi.**WSGIContext**(*wsgi_app*)

Bases: object

This class provides a means to provide context (scope) for a middleware filter to have access to the wsgi start_response results like the request status and headers.

class swift.common.wsgi.**WorkersStrategy**(*conf*, *logger*)

Bases: *swift.common.wsgi.StrategyBase*

WSGI server management strategy object for a single bind port and listen socket shared by a configured number of forked-off workers.

Tracking data is a map of `pid -> socket`.

Used in *run_wsgi()*.

Parameters

- **conf** (*dict*) Server configuration dictionary.
- **logger** The servers `LogAdaptor` object.

iter_sockets()

Yields all known listen sockets.

log_sock_exit(*sock*, *_unused*)

Log a servers exit.

Parameters

- **sock** (*socket*) The listen socket for the worker just started.
- **_unused** The sockets opaque_data yielded by *new_worker_socks()*.

loop_timeout()

We want to keep from busy-waiting, but we also need a non-None value so the main loop gets a chance to tell whether it should keep running or not (e.g. SIGHUP received).

So we return 0.5.

new_worker_socks()

Yield a sequence of (socket, opqaue_data) tuples for each server which should be forked-off and started.

The opaque_data item for each socket will passed into the *log_sock_exit()* and *register_worker_start()* methods where it will be ignored.

no_fork_sock()

Return a server listen socket if the server should run in the foreground (no fork).

register_worker_exit(*pid*)

Called when a worker has exited.

NOTE: a re-execed server can reap the dead worker PIDs from the old server process that is being replaced as part of a service reload (SIGUSR1). So we need to be robust to getting some unknown PID here.

Parameters **pid** (*int*) The PID of the worker that exited.

register_worker_start(*sock*, *_unused*, *pid*)

Called when a new worker is started.

Parameters

- **sock** (*socket*) The listen socket for the worker just started.
- **_unused** The sockets opaque_data yielded by new_worker_socks().
- **pid** (*int*) The new worker process PID

swift.common.wsgi.**get_socket**(*conf*)

Bind socket to bind ip:port in conf

Parameters **conf** Configuration dict to read settings from

Returns a socket object as returned from socket.listen or ssl.wrap_socket if conf specifies cert_file

swift.common.wsgi.**init_request_processor**(*conf_path*, *app_section*, **args*, ***kwargs*)

Loads common settings from conf Sets the logger Loads the request processor

Parameters

- **conf_path** Path to paste.deploy style configuration file/directory
- **app_section** App name from conf file to load config from

Returns the loaded application entry point

Raises *ConfigFileError* Exception is raised for config file error

swift.common.wsgi.**load_app_config**(*conf_file*)

Read the app config section from a config file.

Parameters **conf_file** path to a config file

Returns a dict

swift.common.wsgi.**loadapp**(*conf_file*, *global_conf=None*, *allow_modify_pipeline=True*)

Loads a context from a config file, and if the context is a pipeline then presents the app with the opportunity to modify the pipeline.

Parameters

- **conf_file** path to a config file
- **global_conf** a dict of options to update the loaded config. Options in `global_conf` will override those in `conf_file` except where the `conf_file` option is preceded by `set`.
- **allow_modify_pipeline** if True, and the context is a pipeline, and the loaded app has a `modify_wsgi_pipeline` property, then that property will be called before the pipeline is loaded.

Returns the loaded app

swift.common.wsgi.**make_env**(*env*, *method=None*, *path=None*, *agent='Swift'*, *query_string=None*, *swift_source=None*)

Returns a new fresh WSGI environment.

Parameters

- **env** The WSGI environment to base the new environment on.
- **method** The new REQUEST_METHOD or None to use the original.
- **path** The new path_info or none to use the original. path should NOT be quoted. When building a url, a Webob Request (in accordance with wsgi spec) will quote env[PATH_INFO]. url += quote(environ[PATH_INFO])
- **query_string** The new query_string or none to use the original. When building a url, a Webob Request will append the query string directly to the url. url += ? + env[QUERY_STRING]
- **agent** The HTTP user agent to use; default Swift. You can put %(orig)s in the agent to have it replaced with the original envs HTTP_USER_AGENT, such as %(orig)s StaticWeb. You also set agent to None to use the original envs HTTP_USER_AGENT or to have no HTTP_USER_AGENT.
- **swift_source** Used to mark the request as originating out of middleware. Will be logged in proxy logs.

Returns Fresh WSGI environment.

swift.common.wsgi.**make_pre_authed_env**(*env*, *method=None*, *path=None*, *agent='Swift'*, *query_string=None*, *swift_source=None*)

Same as *make_env()* but with preauthorization.

swift.common.wsgi.**make_pre_authed_request**(*env*, *method=None*, *path=None*, *body=None*, *headers=None*, *agent='Swift'*, *swift_source=None*)

Same as *make_subrequest()* but with preauthorization.

swift.common.wsgi.**make_subrequest**(*env*, *method=None*, *path=None*, *body=None*, *headers=None*, *agent='Swift'*, *swift_source=None*, *make_env=<function make_env>*)

Makes a new swob.Request based on the current env but with the parameters specified.

Parameters

- **env** The WSGI environment to base the new request on.
- **method** HTTP method of new request; default is from the original env.
- **path** HTTP path of new request; default is from the original env. path should be compatible with what you would send to Request.blank. path should be quoted and it can include a query string. for example: /a%20space?unicode_str%E8%AA%9E=y%20es
- **body** HTTP body of new request; empty by default.
- **headers** Extra HTTP headers of new request; None by default.
- **agent** The HTTP user agent to use; default Swift. You can put %(orig)s in the agent to have it replaced with the original envs HTTP_USER_AGENT, such as %(orig)s StaticWeb. You also set agent to None to use the original envs HTTP_USER_AGENT or to have no HTTP_USER_AGENT.
- **swift_source** Used to mark the request as originating out of middleware. Will be logged in proxy logs.
- **make_env** make_subrequest calls this make_env to help build the swob.Request.

Returns Fresh swob.Request object.

swift.common.wsgi.**run_wsgi**(*conf_path*, *app_section*, **args*, ***kwargs*)

Runs the server according to some strategy. The default strategy runs a specified number of workers in pre-fork model. The object-server (only) may use a servers-per-port strategy if its config has a servers_per_port setting with a value greater than zero.

Parameters

- **conf_path** Path to paste.deploy style configuration file/directory
- **app_section** App name from conf file to load config from
- **allow_modify_pipeline** Boolean for whether the server should have an opportunity to change its own pipeline. Defaults to True

Returns 0 if successful, nonzero otherwise

swift.common.wsgi.**wrap_conf_type**(*f*)

Wrap a function whos first argument is a paste.deploy style config uri, such that you can pass it an un-adorned raw filesystem path (or config string) and the config directive (either config:, config_dir:, or config_str:) will be added automatically based on the type of entity (either a file or directory, or if no such entity on the file system - just a string) before passing it through to the paste.deploy function.

9.7.16 Storage Policy

class swift.common.storage_policy.**BaseStoragePolicy**(*idx*, *name=''*, *is_default=False*, *is_deprecated=False*, *object_ring=None*, *aliases=''*, *diskfile_module='egg:swift#replication.fs'*)

Bases: object

Represents a storage policy. Not meant to be instantiated directly; implement a derived subclasses (e.g. StoragePolicy, ECStoragePolicy, etc) or use *reload_storage_policies()* to load POLICIES from swift.conf.

The object_ring property is lazy loaded once the services swift_dir is known via *get_object_ring()*, but it may be over-ridden via object_ring kwarg at create time for testing or actively loaded with load_ring().

add_name(*name*)

Adds an alias name to the storage policy. Shouldnt be called directly from the storage policy but instead through the storage policy collection class, so lookups by name resolve correctly.

Parameters **name** a new alias for the storage policy

change_primary_name(*name*)

Changes the primary/default name of the policy to a specified name.

Parameters **name** a string name to replace the current primary name.

get_diskfile_manager(**args*, ***kwargs*)

Return an instance of the diskfile manager class configured for this storage policy.

Parameters

- **args** positional args to pass to the diskfile manager constructor.
- **kwargs** keyword args to pass to the diskfile manager constructor.

Returns A disk file manager instance.

get_info(*config=False*)

Return the info dict and conf file options for this policy.

Parameters **config** boolean, if True all config options are returned

load_ring(*swift_dir*, *reload_time=None*)

Load the ring for this policy immediately.

Parameters

- **swift_dir** path to rings

- **reload_time** time interval in seconds to check for a ring change

property quorum

Number of successful backend requests needed for the proxy to consider the client request successful.

classmethod register(*policy_type*)

Decorator for Storage Policy implementations to register their StoragePolicy class. This will also set the policy_type attribute on the registered implementation.

remove_name(*name*)

Removes an alias name from the storage policy. Shouldnt be called directly from the storage policy but instead through the storage policy collection class, so lookups by name resolve correctly. If the name removed is the primary name then the next available alias will be adopted as the new primary name.

Parameters **name** a name assigned to the storage policy

validate_ring_data(*ring_data*)

Validation hook used when loading the ring; currently only used for EC

class swift.common.storage_policy.**ECStoragePolicy**(*idx*, *name=''*, *aliases=''*, *is_default=False*, *is_deprecated=False*, *object_ring=None*, *diskfile_module='egg:swift#erasure_coding.fs'*, *ec_segment_size=1048576*, *ec_type=None*, *ec_ndata=None*, *ec_nparity=None*, *ec_duplication_factor=1*)

Bases: *swift.common.storage_policy.BaseStoragePolicy*

Represents a storage policy of type erasure_coding.

Not meant to be instantiated directly; use *reload_storage_policies()* to load POLICIES from swift.conf.

property ec_scheme_description

This short hand form of the important parts of the ec schema is stored in Object System Metadata on the EC Fragment Archives for debugging.

property fragment_size

Maximum length of a fragment, including header.

NB: a fragment archive is a sequence of 0 or more max-length fragments followed by one possibly-shorter fragment.

get_backend_index(*node_index*)

Backend index for PyECLib

Parameters **node_index** integer of node index

Returns integer of actual fragment index. if param is not an integer, return None instead

get_info(*config=False*)

Return the info dict and conf file options for this policy.

Parameters **config** boolean, if True all config options are returned

property quorum

Number of successful backend requests needed for the proxy to consider the client PUT request successful.

The quorum size for EC policies defines the minimum number of data + parity elements required to be able to guarantee the desired fault tolerance, which is the number of data elements supplemented by the minimum number of parity elements required by the chosen erasure coding scheme.

For example, for Reed-Solomon, the minimum number parity elements required is 1, and thus the quorum_size requirement is ec_ndata + 1.

Given the number of parity elements required is not the same for every erasure coding scheme, consult PyECLib for min_parity_fragments_needed()

validate_ring_data(*ring_data*)

EC specific validation

Replica count check - we need _at_least_ (#data + #parity) replicas configured. Also if the replica count is larger than exactly that number theres a non-zero risk of error for code that is considering the number of nodes in the primary list from the ring.

exception swift.common.storage_policy.**PolicyError**(*msg*, *index=None*)

Bases: ValueError

class swift.common.storage_policy.**StoragePolicy**(*idx*, *name=''*, *is_default=False*, *is_deprecated=False*, *object_ring=None*, *aliases=''*, *diskfile_module='egg:swift#replication.fs'*)

Bases: *swift.common.storage_policy.BaseStoragePolicy*

Represents a storage policy of type replication. Default storage policy class unless otherwise overridden from swift.conf.

Not meant to be instantiated directly; use *reload_storage_policies()* to load POLICIES from swift.conf.

property quorum

Quorum concept in the replication case: floor(number of replica / 2) + 1

class swift.common.storage_policy.**StoragePolicyCollection**(*pols*)

Bases: object

This class represents the collection of valid storage policies for the cluster and is instantiated as *StoragePolicy* objects are added to the collection when swift.conf is parsed by *parse_storage_policies()*.

When a StoragePolicyCollection is created, the following validation is enforced:

- If a policy with index 0 is not declared and no other policies defined, Swift will create one
- The policy index must be a non-negative integer
- If no policy is declared as the default and no other policies are defined, the policy with index 0 is set as the default

- Policy indexes must be unique
- Policy names are required
- Policy names are case insensitive
- Policy names must contain only letters, digits or a dash
- Policy names must be unique
- The policy name Policy-0 can only be used for the policy with index 0
- If any policies are defined, exactly one policy must be declared default
- Deprecated policies can not be declared the default

`add_policy_alias`(*policy_index*, **aliases*)

Adds a new name or names to a policy

Parameters

- **policy_index** index of a policy in this policy collection.
- **aliases** arbitrary number of string policy names to add.

`change_policy_primary_name`(*policy_index*, *new_name*)

Changes the primary or default name of a policy. The new primary name can be an alias that already belongs to the policy or a completely new name.

Parameters

- **policy_index** index of a policy in this policy collection.
- **new_name** a string name to set as the new default name.

`get_by_index`(*index*)

Find a storage policy by its index.

An index of None will be treated as 0.

Parameters **index** numeric index of the storage policy

Returns storage policy, or None if no such policy

`get_by_name`(*name*)

Find a storage policy by its name.

Parameters **name** name of the policy

Returns storage policy, or None

`get_object_ring`(*policy_idx*, *swift_dir*)

Get the ring object to use to handle a request based on its policy.

An index of None will be treated as 0.

Parameters

- **policy_idx** policy index as defined in swift.conf
- **swift_dir** swift_dir used by the caller

Returns appropriate ring object

get_policy_info()

Build info about policies for the /info endpoint

Returns list of dicts containing relevant policy information

remove_policy_alias(**aliases*)

Removes a name or names from a policy. If the name removed is the primary name then the next available alias will be adopted as the new primary name.

Parameters **aliases** arbitrary number of existing policy names to remove.

class swift.common.storage_policy.**StoragePolicySingleton**

Bases: object

An instance of this class is the primary interface to storage policies exposed as a module level global named POLICIES. This global reference wraps _POLICIES which is normally instantiated by parsing swift.conf and will result in an instance of *StoragePolicyCollection*.

You should never patch this instance directly, instead patch the module level _POLICIES instance so that swift code which imported POLICIES directly will reference the patched *StoragePolicyCollection*.

swift.common.storage_policy.**get_policy_string**(*base*, *policy_or_index*)

Helper function to construct a string from a base and the policy. Used to encode the policy index into either a file name or a directory name by various modules.

Parameters

- **base** the base string
- **policy_or_index** StoragePolicy instance, or an index (string or int), if None the legacy storage Policy-0 is assumed.

Returns base name with policy index added

Raises *PolicyError* if no policy exists with the given policy_index

swift.common.storage_policy.**parse_storage_policies**(*conf*)

Parse storage policies in swift.conf - note that validation is done when the *StoragePolicyCollection* is instantiated.

Parameters **conf** ConfigParser parser object for swift.conf

swift.common.storage_policy.**reload_storage_policies**()

Reload POLICIES from swift.conf.

swift.common.storage_policy.**split_policy_string**(*policy_string*)

Helper function to convert a string representing a base and a policy. Used to decode the policy from either a file name or a directory name by various modules.

Parameters **policy_string** base name with policy index added

Raises *PolicyError* if given index does not map to a valid policy

Returns a tuple, in the form (base, policy) where base is the base string and policy is the StoragePolicy instance for the index encoded in the policy_string.

9.8 Middleware

9.8.1 Account Quotas

`account_quotas` is a middleware which blocks write requests (PUT, POST) if a given account quota (in bytes) is exceeded while DELETE requests are still allowed.

`account_quotas` uses the `x-account-meta-quota-bytes` metadata entry to store the quota. Write requests to this metadata entry are only permitted for resellers. There is no quota limit if `x-account-meta-quota-bytes` is not set.

The `account_quotas` middleware should be added to the pipeline in your `/etc/swift/proxy-server.conf` file just after any auth middleware. For example:

```
[pipeline:main]
pipeline = catch_errors cache tempauth account_quotas proxy-server

[filter:account_quotas]
use = egg:swift#account_quotas
```

To set the quota on an account:

```
swift -A http://127.0.0.1:8080/auth/v1.0 -U account:reseller -K secret post -
↪m quota-bytes:10000
```

Remove the quota:

```
swift -A http://127.0.0.1:8080/auth/v1.0 -U account:reseller -K secret post -
↪m quota-bytes:
```

The same limitations apply for the account quotas as for the container quotas.

For example, when uploading an object without a content-length header the proxy server doesnt know the final size of the currently uploaded object and the upload will be allowed if the current account size is within the quota. Due to the eventual consistency further uploads might be possible until the account size has been updated.

class swift.common.middleware.account_quotas.**AccountQuotaMiddleware**(*app*, **args*, ***kwargs*)

Bases: object

Account quota middleware

See above for a full description.

swift.common.middleware.account_quotas.**filter_factory**(*global_conf*, ***local_conf*)

Returns a WSGI filter app for use with paste.deploy.

9.8.2 AWS S3 Api

The s3api middleware will emulate the S3 REST api on top of swift.

To enable this middleware to your configuration, add the s3api middleware in front of the auth middleware. See `proxy-server.conf-sample` for more detail and configurable options.

To set up your client, ensure you are using the tempauth or keystone auth system for swift project. When your swift on a SAIO environment, make sure you have setting the tempauth middleware configuration in `proxy-server.conf`, and the access key will be the concatenation of the account and user strings that should look like test:tester, and the secret access key is the account password. The host should also point to the swift storage hostname.

The tempauth option example:

```
[filter:tempauth]
use = egg:swift#tempauth
user_admin_admin = admin .admin .reseller_admin
user_test_tester = testing
```

An example client using tempauth with the python boto library is as follows:

```
from boto.s3.connection import S3Connection
connection = S3Connection(
    aws_access_key_id='test:tester',
    aws_secret_access_key='testing',
    port=8080,
    host='127.0.0.1',
    is_secure=False,
    calling_format=boto.s3.connection.OrdinaryCallingFormat())
```

And if you using keystone auth, you need the ec2 credentials, which can be downloaded from the API Endpoints tab of the dashboard or by openstack ec2 command.

Here is showing to create an EC2 credential:

```
# openstack ec2 credentials create
+------------+-----------------------------------------------------+
| Field      | Value                                               |
+------------+-----------------------------------------------------+
| access     | c2e30f2cd5204b69a39b3f1130ca8f61                    |
| links      | {u'self': u'http://controller:5000/v3/......'}      |
| project_id | 407731a6c2d0425c86d1e7f12a900488                    |
| secret     | baab242d192a4cd6b68696863e07ed59                    |
| trust_id   | None                                                |
| user_id    | 00f0ee06afe74f81b410f3fe03d34fbc                    |
+------------+-----------------------------------------------------+
```

An example client using keystone auth with the python boto library will be:

```
from boto.s3.connection import S3Connection
connection = S3Connection(
    aws_access_key_id='c2e30f2cd5204b69a39b3f1130ca8f61',
```

(continues on next page)

(continued from previous page)

```
    aws_secret_access_key='baab242d192a4cd6b68696863e07ed59',
    port=8080,
    host='127.0.0.1',
    is_secure=False,
    calling_format=boto.s3.connection.OrdinaryCallingFormat())
```

Deployment

Proxy-Server Setting

Set s3api before your auth in your pipeline in `proxy-server.conf` file. To enable all compatibility currently supported, you should make sure that bulk, slo, and your auth middleware are also included in your proxy pipeline setting.

Using tempauth, the minimum example config is:

```
[pipeline:main]
pipeline = proxy-logging cache s3api tempauth bulk slo proxy-logging proxy-
↪server
```

When using keystone, the config will be:

```
[pipeline:main]
pipeline = proxy-logging cache authtoken s3api s3token keystoneauth bulk slo␣
↪proxy-logging proxy-server
```

Finally, add the s3api middleware section:

```
[filter:s3api]
use = egg:swift#s3api
```

Note: `keystonemiddleware.authtoken` can be located before/after s3api but we recommend to put it before s3api because when authtoken is after s3api, both authtoken and s3token will issue the acceptable token to keystone (i.e. authenticate twice). And in the `keystonemiddleware.authtoken` middleware , you should set `delay_auth_decision` option to `True`.

Constraints

Currently, the s3api is being ported from https://github.com/openstack/swift3 so any existing issues in swift3 are still remaining. Please make sure descriptions in the example `proxy-server.conf` and what happens with the config, before enabling the options.

Supported API

The compatibility will continue to be improved upstream, you can keep and eye on compatibility via a check tool build by SwiftStack. See https://github.com/swiftstack/s3compat in detail.

class swift.common.middleware.s3api.s3api.**S3ApiMiddleware**(*app*, *wsgi_conf*, **args*, ***kwargs*)

Bases: object

S3Api: S3 compatibility middleware

check_filter_order(*pipeline*, *required_filters*)

Check that required filters are present in order in the pipeline.

check_pipeline(*wsgi_conf*)

Check that proxy-server.conf has an appropriate pipeline for s3api.

swift.common.middleware.s3api.s3api.**filter_factory**(*global_conf*, ***local_conf*)

Standard filter factory to use the middleware with paste.deploy

S3 Token Middleware

s3token middleware is for authentication with s3api + keystone. This middleware:

- Gets a request from the s3api middleware with an S3 Authorization access key.
- Validates s3 token with Keystone.
- Transforms the account name to AUTH_%(tenant_name).
- Optionally can retrieve and cache secret from keystone to validate signature locally

---

Note: If upgrading from swift3, the auth_version config option has been removed, and the auth_uri option now includes the Keystone API version. If you previously had a configuration like

```
[filter:s3token]
use = egg:swift3#s3token
auth_uri = https://keystonehost:35357
auth_version = 3
```

you should now use

```
[filter:s3token]
use = egg:swift#s3token
auth_uri = https://keystonehost:35357/v3
```

---

class swift.common.middleware.s3api.s3token.**S3Token**(*app*, *conf*)

Bases: object

Middleware that handles S3 authentication.

swift.common.middleware.s3api.s3token.**filter_factory**(*global_conf*, ***local_conf*)

Returns a WSGI filter app for use with paste.deploy.

class swift.common.middleware.s3api.s3request.**HashingInput**(*reader*, *content_length*, *hasher*, *expected_hex_hash*)

Bases: object

wsgi.input wrapper to verify the hash of the input as its read.

class swift.common.middleware.s3api.s3request.**S3AclRequest**(*env*, *app=None*, *conf=None*)

Bases: *swift.common.middleware.s3api.s3request.S3Request*

S3Acl request object.

authenticate(*app*)

authenticate method will run pre-authenticate request and retrieve account information. Note that it currently supports only keystone and tempauth. (no support for the third party authentication middleware)

get_acl_response(*app*, *method=None*, *container=None*, *obj=None*, *headers=None*, *body=None*, *query=None*)

Wrapper method of _get_response to add s3 acl information from response sysmeta headers.

get_response(*app*, *method=None*, *container=None*, *obj=None*, *headers=None*, *body=None*, *query=None*)

Wrap up get_response call to hook with acl handling method.

to_swift_req(*method*, *container*, *obj*, *query=None*, *body=None*, *headers=None*)

Create a Swift request based on this requests environment.

class swift.common.middleware.s3api.s3request.**S3Request**(*env*, *app=None*, *conf=None*)

Bases: *swift.common.swob.Request*

S3 request object.

property body

swob.Request.body is not secure against malicious input. It consumes too much memory without any check when the request body is excessively large. Use xml() instead.

property bucket_acl

Get and set the container acl property

check_copy_source(*app*)

check_copy_source checks the copy source existence and if copying an object to itself, for illegal request parameters

Returns the source HEAD response

get_container_info(*app*)

get_container_info will return a result dict of get_container_info from the backend Swift.

Returns a dictionary of container info from swift.controllers.base.get_container_info

Raises NoSuchBucket when the container doesnt exist

Raises InternalError when the request failed without 404

get_response(*app*, *method=None*, *container=None*, *obj=None*, *headers=None*, *body=None*, *query=None*)

get_response is an entry point to be extended for child classes. If additional tasks needed at that time of getting swift response, we can override this method. swift.common.middleware.s3api.s3request.S3Request need to just call _get_response to get pure swift response.

property object_acl

Get and set the object acl property

property timestamp

S3Timestamp from Date header. If X-Amz-Date header specified, it will be prior to Date header.

:return : S3Timestamp instance

to_swift_req(*method*, *container*, *obj*, *query=None*, *body=None*, *headers=None*)

Create a Swift request based on this requests environment.

xml(*max_length*)

Similar to swob.Request.body, but it checks the content length before creating a body string.

class swift.common.middleware.s3api.s3request.**SigV4Mixin**

Bases: object

A request class mixin to provide S3 signature v4 functionality

property timestamp

Return timestamp string according to the auth type The difference from v2 is v4 have to see X-Amz-Date even though its query auth type.

class swift.common.middleware.s3api.s3request.**SigV4Request**(*env*, *app=None*, *conf=None*)

Bases: *swift.common.middleware.s3api.s3request.SigV4Mixin*, *swift.common.middleware.s3api.s3request.S3Request*

class swift.common.middleware.s3api.s3request.**SigV4S3AclRequest**(*env*, *app=None*, *conf=None*)

Bases: *swift.common.middleware.s3api.s3request.SigV4Mixin*, *swift.common.middleware.s3api.s3request.S3AclRequest*

swift.common.middleware.s3api.s3request.**get_request_class**(*env*, *s3_acl*)

Helper function to find a request class to use from Map

exception swift.common.middleware.s3api.s3response.**AccessDenied**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**AccountProblem**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**AmbiguousGrantByEmailAddress**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**AuthorizationHeaderMalformed**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**AuthorizationQueryParametersError**(*msg=No*
**args*,
***kwarg*

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**BadDigest**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**BrokenMPU**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**BucketAlreadyExists**(*bucket*, *msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**BucketAlreadyOwnedByYou**(*bucket*, *msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**BucketNotEmpty**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**CredentialsNotSupported**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**CrossLocationLoggingProhibited**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**EntityTooLarge**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**EntityTooSmall**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**ErrorResponse**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.S3ResponseBase*, *swift.common.swob.HTTPException*

S3 error object.

Reference information about S3 errors is available at: http://docs.aws.amazon.com/AmazonS3/latest/API/ErrorResponses.html

exception swift.common.middleware.s3api.s3response.**ExpiredToken**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

class swift.common.middleware.s3api.s3response.**HeaderKeyDict**(*base_headers=None*, ***kwargs*)

Bases: swift.common.header_key_dict.HeaderKeyDict

Similar to the Swifts normal HeaderKeyDict class, but its key name is normalized as S3 clients expect.

exception swift.common.middleware.s3api.s3response.**IllegalVersioningConfigurationException**(

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**IncompleteBody**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**IncorrectNumberOfFilesInPostRequest**(*msg=*

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InlineDataTooLarge**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InternalError**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidAccessKeyId**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidArgument**(*name*, *value*, *msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidBucketName**(*bucket*, *msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidBucketState**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidDigest**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidLocationConstraint**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidObjectState**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidPart**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidPartOrder**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidPayer**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidPolicyDocument**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidRange**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidRequest**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidSOAPRequest**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidSecurity**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidStorageClass**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidTargetBucketForLogging**(*bucket*, *msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidToken**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**InvalidURI**(*uri*, *msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**KeyTooLongError**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**MalformedACLError**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**MalformedPOSTRequest**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**MalformedXML**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**MaxMessageLengthExceeded**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**MaxPostPreDataLengthExceededError**(*msg=No*
**args*,
***kwarg*

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**MetadataTooLarge**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**MethodNotAllowed**(*method*, *resource_type*, *msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**MissingContentLength**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**MissingRequestBodyError**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**MissingSecurityElement**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**MissingSecurityHeader**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**NoLoggingStatusForKey**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**NoSuchBucket**(*bucket*, *msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**NoSuchKey**(*key*, *msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**NoSuchLifecycleConfiguration**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**NoSuchUpload**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**NoSuchVersion**(*key*, *version_id*, *msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**NotSignedUp**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**NotSuchBucketPolicy**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**OperationAborted**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**PermanentRedirect**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**PreconditionFailed**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**Redirect**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**RequestIsNotMultiPartContent**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**RequestTimeTooSkewed**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**RequestTimeout**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**RequestTorrentOfBucketError**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**RestoreAlreadyInProgress**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**S3NotImplemented**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

class swift.common.middleware.s3api.s3response.**S3Response**(**args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.S3ResponseBase*, *swift.common.swob.Response*

Similar to the Response class in Swift, but uses our HeaderKeyDict for headers instead of Swifts HeaderKeyDict. This also translates Swift specific headers to S3 headers.

classmethod from_swift_resp(*sw_resp*)

Create a new S3 response object based on the given Swift response.

class swift.common.middleware.s3api.s3response.**S3ResponseBase**

Bases: object

Base class for swift3 responses.

exception swift.common.middleware.s3api.s3response.**ServiceUnavailable**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**SignatureDoesNotMatch**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**SlowDown**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**TemporaryRedirect**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**TokenRefreshRequired**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**TooManyBuckets**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**UnexpectedContent**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**UnresolvableGrantByEmailAddress**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**UserKeyMustBeSpecified**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.ErrorResponse*

exception swift.common.middleware.s3api.s3response.**VersionedBucketNotEmpty**(*msg=None*, **args*, ***kwargs*)

Bases: *swift.common.middleware.s3api.s3response.BucketNotEmpty*

exception swift.common.middleware.s3api.exception.**ACLError**

Bases: *swift.common.middleware.s3api.exception.S3Exception*

exception swift.common.middleware.s3api.exception.**BadSwiftRequest**

Bases: *swift.common.middleware.s3api.exception.S3Exception*

exception swift.common.middleware.s3api.exception.**InvalidSubresource**(*resource*, *cause*)

Bases: *swift.common.middleware.s3api.exception.S3Exception*

exception swift.common.middleware.s3api.exception.**NotS3Request**

Bases: *swift.common.middleware.s3api.exception.S3Exception*

exception swift.common.middleware.s3api.exception.**S3Exception**

Bases: Exception

class swift.common.middleware.s3api.etree.**_Element**(**args*, ***kwargs*)

Bases: lxml.etree.ElementBase

Wrapper Element class of lxml.etree.Element to support a utf-8 encoded non-ascii string as a text.

Why we need this?: Original lxml.etree.Element supports only unicode for the text. It declines maintainability because we have to call a lot of encode/decode methods to apply account/container/object name (i.e. PATH_INFO) to each Element instance. When using this class, we can remove such a redundant codes from swift.common.middleware.s3api middleware.

property text

utf-8 wrapper property of lxml.etree.Element.text

class swift.common.middleware.s3api.utils.**Config**(*base=None*)

Bases: dict

update([*E*], ***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

class swift.common.middleware.s3api.utils.**S3Timestamp**(*timestamp*, *offset=0*, *delta=0*, *check_bounds=True*)

Bases: *swift.common.utils.Timestamp*

property amz_date_format

this format should be like YYYYMMDDThhmmssZ

swift.common.middleware.s3api.utils.**mktime**(*timestamp_str*, *time_format='%Y-%m-%dT%H:%M:%S'*)

mktime creates a float instance in epoch time really like as time.mktime

the difference from time.mktime is allowing to 2 formats string for the argument for the S3 testing usage. TODO: support

Parameters

- **timestamp_str** a string of timestamp formatted as (a) RFC2822 (e.g. date header) (b) %Y-%m-%dT%H:%M:%S (e.g. copy result)
- **time_format** a string of format to parse in (b) process

Returns a float instance in epoch time

swift.common.middleware.s3api.utils.**sysmeta_header**(*resource*, *name*)

Returns the system metadata header for given resource type and name.

swift.common.middleware.s3api.utils.**sysmeta_prefix**(*resource*)

Returns the system metadata prefix for given resource type.

swift.common.middleware.s3api.utils.**validate_bucket_name**(*name*, *dns_compliant_bucket_names*)

Validates the name of the bucket against S3 criteria, http://docs.amazonwebservices.com/AmazonS3/latest/BucketRestrictions.html True is valid, False is invalid.

s3apis ACLs implementation

s3api uses a different implementation approach to achieve S3 ACLs.

First, we should understand what we have to design to achieve real S3 ACLs. Current s3api(real S3)s ACLs Model is as follows:

```
AccessControlPolicy:
    Owner:
    AccessControlList:
```

(continues on next page)

(continued from previous page)

```
        Grant[n]:
            (Grantee, Permission)
```

Each bucket or object has its own acl consisting of Owner and AcessControlList. AccessControlList can contain some Grants. By default, AccessControlList has only one Grant to allow FULL CONTROLL to owner. Each Grant includes single pair with Grantee, Permission. Grantee is the user (or user group) allowed the given permission.

This module defines the groups and the relation tree.

If you wanna get more information about S3s ACLs model in detail, please see official documentation here,

http://docs.aws.amazon.com/AmazonS3/latest/dev/acl-overview.html

class swift.common.middleware.s3api.subresource.**ACL**(*owner*, *grants=None*, *s3_acl=False*, *allow_no_owner=False*)

Bases: object

S3 ACL class.

Refs (S3 API - acl-overview: http://docs.aws.amazon.com/AmazonS3/latest/dev/acl-overview.html):

The sample ACL includes an Owner element identifying the owner via the AWS accounts canonical user ID. The Grant element identifies the grantee (either an AWS account or a predefined group), and the permission granted. This default ACL has one Grant element for the owner. You grant permissions by adding Grant elements, each grant identifying the grantee and the permission.

check_owner(*user_id*)

Check that the user is an owner.

check_permission(*user_id*, *permission*)

Check that the user has a permission.

elem()

Decode the value to an ACL instance.

classmethod from_elem(*elem*, *s3_acl=False*, *allow_no_owner=False*)

Convert an ElementTree to an ACL instance

classmethod from_headers(*headers*, *bucket_owner*, *object_owner=None*, *as_private=True*)

Convert HTTP headers to an ACL instance.

class swift.common.middleware.s3api.subresource.**AllUsers**

Bases: *swift.common.middleware.s3api.subresource.Group*

Access permission to this group allows anyone to access the resource. The requests can be signed (authenticated) or unsigned (anonymous). Unsigned requests omit the Authentication header in the request.

Note: s3api regards unsigned requests as Swift API accesses, and bypasses them to Swift. As a result, AllUsers behaves completely same as AuthenticatedUsers.

class swift.common.middleware.s3api.subresource.**AuthenticatedUsers**

Bases: *swift.common.middleware.s3api.subresource.Group*

This group represents all AWS accounts. Access permission to this group allows any AWS account to access the resource. However, all requests must be signed (authenticated).

class swift.common.middleware.s3api.subresource.**CannedACL**

Bases: object

A dict-like object that returns canned ACL.

class swift.common.middleware.s3api.subresource.**Grant**(*grantee*, *permission*)

Bases: object

Grant Class which includes both Grantee and Permission

elem()

Create an etree element.

classmethod from_elem(*elem*)

Convert an ElementTree to an ACL instance

class swift.common.middleware.s3api.subresource.**Grantee**

Bases: object

Base class for grantee.

Methods:

- init: create a Grantee instance
- elem: create an ElementTree from itself

Static Methods:

- **from_header: convert a grantee string in the HTTP header** to an Grantee instance.
- from_elem: convert a ElementTree to an Grantee instance.

elem()

Get an etree element of this instance.

static from_header(*grantee*)

Convert a grantee string in the HTTP header to an Grantee instance.

class swift.common.middleware.s3api.subresource.**Group**

Bases: *swift.common.middleware.s3api.subresource.Grantee*

Base class for Amazon S3 Predefined Groups

elem()

Get an etree element of this instance.

class swift.common.middleware.s3api.subresource.**LogDelivery**

Bases: *swift.common.middleware.s3api.subresource.Group*

WRITE and READ_ACP permissions on a bucket enables this group to write server access logs to the bucket.

class swift.common.middleware.s3api.subresource.**Owner**(*id*, *name*)

Bases: object

Owner class for S3 accounts

class swift.common.middleware.s3api.subresource.**User**(*name*)

Bases: *swift.common.middleware.s3api.subresource.Grantee*

Canonical user class for S3 accounts.

elem()

Get an etree element of this instance.

swift.common.middleware.s3api.subresource.**canned_acl_grantees**(*bucket_owner*, *object_owner=None*)

A set of predefined grants supported by AWS S3.

swift.common.middleware.s3api.subresource.**decode_acl**(*resource*, *headers*, *allow_no_owner*)

Decode Swift metadata to an ACL instance.

Given a resource type and HTTP headers, this method returns an ACL instance.

swift.common.middleware.s3api.subresource.**encode_acl**(*resource*, *acl*)

Encode an ACL instance to Swift metadata.

Given a resource type and an ACL instance, this method returns HTTP headers, which can be used for Swift metadata.

swift.common.middleware.s3api.subresource.**get_group_subclass_from_uri**(*uri*)

Convert a URI to one of the predefined groups.

Acl Handlers

Why do we need this

To make controller classes clean, we need these handlers. It is really useful for customizing acl checking algorithms for each controller.

Basic Information

BaseAclHandler wraps basic Acl handling. (i.e. it will check acl from ACL_MAP by using HEAD)

How to extend

Make a handler with the name of the controller. (e.g. BucketAclHandler is for BucketController) It consists of method(s) for actual S3 method on controllers as follows.

Example:

```
class BucketAclHandler(BaseAclHandler):
    def PUT:
        << put acl handling algorithms here for PUT bucket >>
```

Note: If the method DONT need to recall _get_response in outside of acl checking, the method have to return the response it needs at the end of method.

class swift.common.middleware.s3api.acl_handlers.**BaseAclHandler**(*req*, *logger*, *container=None*, *obj=None*, *headers=None*)

Bases: object

BaseAclHandler: Handling ACL for basic requests mapped on ACL_MAP

get_acl(*headers*, *body*, *bucket_owner*, *object_owner=None*)

Get ACL instance from S3 (e.g. x-amz-grant) headers or S3 acl xml body.

class swift.common.middleware.s3api.acl_handlers.**BucketAclHandler**(*req*, *logger*, *container=None*, *obj=None*, *headers=None*)

Bases: *swift.common.middleware.s3api.acl_handlers.BaseAclHandler*

BucketAclHandler: Handler for BucketController

class swift.common.middleware.s3api.acl_handlers.**MultiObjectDeleteAclHandler**(*req*, *logger*, *container=None*, *obj=None*, *headers=None*)

Bases: *swift.common.middleware.s3api.acl_handlers.BaseAclHandler*

MultiObjectDeleteAclHandler: Handler for MultiObjectDeleteController

class swift.common.middleware.s3api.acl_handlers.**MultiUploadAclHandler**(*req*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.acl_handlers.BaseAclHandler*

MultiUpload stuff requires acl checking just once for BASE container so that MultiUploadAclHandler extends BaseAclHandler to check acl only when the verb defined. We should define the verb as the first step to request to backend Swift at incoming request.

Basic Rules:

- BASE container name is always w/o MULTIUPLOAD_SUFFIX
- Any check timing is ok but we should check it as soon as possible.

Controller	Verb	CheckResource	Permission
Part	PUT	Container	WRITE
Uploads	GET	Container	READ
Uploads	POST	Container	WRITE
Upload	GET	Container	READ
Upload	DELETE	Container	WRITE
Upload	POST	Container	WRITE

class swift.common.middleware.s3api.acl_handlers.**ObjectAclHandler**(*req*, *logger*, *container=None*, *obj=None*, *headers=None*)

Bases: *swift.common.middleware.s3api.acl_handlers.BaseAclHandler*

ObjectAclHandler: Handler for ObjectController

class swift.common.middleware.s3api.acl_handlers.**PartAclHandler**(*req*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.acl_handlers.MultiUploadAclHandler*

PartAclHandler: Handler for PartController

class swift.common.middleware.s3api.acl_handlers.**S3AclHandler**(*req*, *logger*, *container=None*, *obj=None*, *headers=None*)

Bases: *swift.common.middleware.s3api.acl_handlers.BaseAclHandler*

S3AclHandler: Handler for S3AclController

class swift.common.middleware.s3api.acl_handlers.**UploadAclHandler**(*req*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.acl_handlers.MultiUploadAclHandler*

UploadAclHandler: Handler for UploadController

class swift.common.middleware.s3api.acl_handlers.**UploadsAclHandler**(*req*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.acl_handlers.MultiUploadAclHandler*

UploadsAclHandler: Handler for UploadsController

swift.common.middleware.s3api.acl_utils.**handle_acl_header**(*req*)

Handle the x-amz-acl header. Note that this header currently used for only normal-acl (not implemented) on s3acl. TODO: add translation to swift acl like as x-container-read to s3acl

swift.common.middleware.s3api.acl_utils.**swift_acl_translate**(*acl*, *group=''*, *user=''*, *xml=False*)

Takes an S3 style ACL and returns a list of header/value pairs that implement that ACL in Swift, or NotImplemented if there isnt a way to do that yet.

class swift.common.middleware.s3api.controllers.base.**Controller**(*app*, *conf*, *logger*, ***kwargs*)

Bases: `object`

Base WSGI controller class for the middleware

classmethod resource_type()

Returns the target resource type of this controller.

class swift.common.middleware.s3api.controllers.base.**UnsupportedController**(*app*, *conf*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.controllers.base.Controller*

Handles unsupported requests.

swift.common.middleware.s3api.controllers.base.**bucket_operation**(*func=None*, *err_resp=None*, *err_msg=None*)

A decorator to ensure that the request is a bucket operation. If the target resource is an object, this decorator updates the request by default so that the controller handles it as a bucket operation. If err_resp is specified, this raises it on error instead.

swift.common.middleware.s3api.controllers.base.**check_container_existence**(*func*)

A decorator to ensure the container existence.

swift.common.middleware.s3api.controllers.base.**object_operation**(*func*)

A decorator to ensure that the request is an object operation. If the target resource is not an object, this raises an error response.

class swift.common.middleware.s3api.controllers.service.**ServiceController**(*app*, *conf*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.controllers.base.Controller*

Handles account level requests.

GET(*req*)

Handle GET Service request

class swift.common.middleware.s3api.controllers.bucket.**BucketController**(*app*, *conf*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.controllers.base.Controller*

Handles bucket request.

DELETE(*req*)

Handle DELETE Bucket request

GET(*req*)

Handle GET Bucket (List Objects) request

HEAD(*req*)

Handle HEAD Bucket (Get Metadata) request

POST(*req*)

Handle POST Bucket request

PUT(*req*)

Handle PUT Bucket request

class swift.common.middleware.s3api.controllers.obj.**ObjectController**(*app*, *conf*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.controllers.base.Controller*

Handles requests on objects

DELETE(*req*)

Handle DELETE Object request

GET(*req*)

Handle GET Object request

HEAD(*req*)

Handle HEAD Object request

PUT(*req*)

Handle PUT Object and PUT Object (Copy) request

class swift.common.middleware.s3api.controllers.acl.**AclController**(*app*, *conf*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.controllers.base.Controller*

Handles the following APIs:

- GET Bucket acl
- PUT Bucket acl
- GET Object acl
- PUT Object acl

Those APIs are logged as ACL operations in the S3 server log.

GET(*req*)

Handles GET Bucket acl and GET Object acl.

PUT(*req*)

Handles PUT Bucket acl and PUT Object acl.

swift.common.middleware.s3api.controllers.acl.**get_acl**(*account_name*, *headers*)

Attempts to construct an S3 ACL based on what is found in the swift headers

class swift.common.middleware.s3api.controllers.s3_acl.**S3AclController**(*app*, *conf*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.controllers.base.Controller*

Handles the following APIs:

- GET Bucket acl
- PUT Bucket acl
- GET Object acl
- PUT Object acl

Those APIs are logged as ACL operations in the S3 server log.

GET(*req*)

Handles GET Bucket acl and GET Object acl.

PUT(*req*)

Handles PUT Bucket acl and PUT Object acl.

Implementation of S3 Multipart Upload.

This module implements S3 Multipart Upload APIs with the Swift SLO feature. The following explains how S3api uses swift container and objects to store S3 upload information:

[bucket]+segments

A container to store upload information. [bucket] is the original bucket where multipart upload is initiated.

[bucket]+segments/[upload_id]

An object of the ongoing upload id. The object is empty and used for checking the target upload status. If the object exists, it means that the upload is initiated but not either completed or aborted.

[bucket]+segments/[upload_id]/[part_number]

The last suffix is the part number under the upload id. When the client uploads the parts, they will be stored in the namespace with [bucket]+segments/[upload_id]/[part_number].

Example listing result in the [bucket]+segments container:

```
[bucket]+segments/[upload_id1]  # upload id object for upload_id1
[bucket]+segments/[upload_id1]/1  # part object for upload_id1
[bucket]+segments/[upload_id1]/2  # part object for upload_id1
[bucket]+segments/[upload_id1]/3  # part object for upload_id1
[bucket]+segments/[upload_id2]  # upload id object for upload_id2
[bucket]+segments/[upload_id2]/1  # part object for upload_id2
[bucket]+segments/[upload_id2]/2  # part object for upload_id2
   .
   .
```

Those part objects are directly used as segments of a Swift Static Large Object when the multipart upload is completed.

class swift.common.middleware.s3api.controllers.multi_upload.**PartController**(*app*, *conf*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.controllers.base.Controller*

Handles the following APIs:

- Upload Part
- Upload Part - Copy

Those APIs are logged as PART operations in the S3 server log.

PUT(*req*)

Handles Upload Part and Upload Part Copy.

class swift.common.middleware.s3api.controllers.multi_upload.**UploadController**(*app*, *conf*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.controllers.base.Controller*

Handles the following APIs:

- List Parts
- Abort Multipart Upload
- Complete Multipart Upload

Those APIs are logged as UPLOAD operations in the S3 server log.

DELETE(*req*)

Handles Abort Multipart Upload.

GET(*req*)

Handles List Parts.

POST(*req*)

Handles Complete Multipart Upload.

class swift.common.middleware.s3api.controllers.multi_upload.**UploadsController**(*app*, *conf*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.controllers.base.Controller*

Handles the following APIs:

- List Multipart Uploads
- Initiate Multipart Upload

Those APIs are logged as UPLOADS operations in the S3 server log.

GET(*req*)

Handles List Multipart Uploads

POST(*req*)

Handles Initiate Multipart Upload.

class swift.common.middleware.s3api.controllers.multi_delete.**MultiObjectDeleteController**(*app*, *conf*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.controllers.base.Controller*

Handles Delete Multiple Objects, which is logged as a MULTI_OBJECT_DELETE operation in the S3 server log.

POST(*req*)

Handles Delete Multiple Objects.

class swift.common.middleware.s3api.controllers.versioning.**VersioningController**(*app*, *conf*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.controllers.base.Controller*

Handles the following APIs:

- GET Bucket versioning
- PUT Bucket versioning

Those APIs are logged as VERSIONING operations in the S3 server log.

GET(*req*)

Handles GET Bucket versioning.

PUT(*req*)

Handles PUT Bucket versioning.

class swift.common.middleware.s3api.controllers.location.**LocationController**(*app*, *conf*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.controllers.base.Controller*

Handles GET Bucket location, which is logged as a LOCATION operation in the S3 server log.

GET(*req*)

Handles GET Bucket location.

class swift.common.middleware.s3api.controllers.logging.**LoggingStatusController**(*app*, *conf*, *logger*, ***kwargs*)

Bases: *swift.common.middleware.s3api.controllers.base.Controller*

Handles the following APIs:

- GET Bucket logging
- PUT Bucket logging

Those APIs are logged as LOGGING_STATUS operations in the S3 server log.

GET(*req*)
Handles GET Bucket logging.

PUT(*req*)
Handles PUT Bucket logging.

9.8.3 Bulk Operations (Delete and Archive Auto Extraction)

Middleware that will perform many operations on a single request.

Extract Archive

Expand tar files into a Swift account. Request must be a PUT with the query parameter ?`extract-archive=format` specifying the format of archive file. Accepted formats are tar, tar.gz, and tar.bz2.

For a PUT to the following url:

```
/v1/AUTH_Account/$UPLOAD_PATH?extract-archive=tar.gz
```

UPLOAD_PATH is where the files will be expanded to. UPLOAD_PATH can be a container, a pseudo-directory within a container, or an empty string. The destination of a file in the archive will be built as follows:

```
/v1/AUTH_Account/$UPLOAD_PATH/$FILE_PATH
```

Where FILE_PATH is the file name from the listing in the tar file.

If the UPLOAD_PATH is an empty string, containers will be auto created accordingly and files in the tar that would not map to any container (files in the base directory) will be ignored.

Only regular files will be uploaded. Empty directories, symlinks, etc will not be uploaded.

Content Type

If the content-type header is set in the extract-archive call, Swift will assign that content-type to all the underlying files. The bulk middleware will extract the archive file and send the internal files using PUT operations using the same headers from the original request (e.g. auth-tokens, content-Type, etc.). Notice that any middleware call that follows the bulk middleware does not know if this was a bulk request or if these were individual requests sent by the user.

In order to make Swift detect the content-type for the files based on the file extension, the content-type in the extract-archive call should not be set. Alternatively, it is possible to explicitly tell Swift to detect the content type using this header:

```
X-Detect-Content-Type: true
```

For example:

```
curl -X PUT http://127.0.0.1/v1/AUTH_acc/cont/$?extract-archive=tar
 -T backup.tar
 -H "Content-Type: application/x-tar"
 -H "X-Auth-Token: xxx"
 -H "X-Detect-Content-Type: true"
```

Assigning Metadata

The tar file format (1) allows for UTF-8 key/value pairs to be associated with each file in an archive. If a file has extended attributes, then tar will store those as key/value pairs. The bulk middleware can read those extended attributes and convert them to Swift object metadata. Attributes starting with user.meta are converted to object metadata, and user.mime_type is converted to Content-Type.

For example:

```
setfattr -n user.mime_type -v "application/python-setup" setup.py
setfattr -n user.meta.lunch -v "burger and fries" setup.py
setfattr -n user.meta.dinner -v "baked ziti" setup.py
setfattr -n user.stuff -v "whee" setup.py
```

Will get translated to headers:

```
Content-Type: application/python-setup
X-Object-Meta-Lunch: burger and fries
X-Object-Meta-Dinner: baked ziti
```

The bulk middleware will handle xattrs stored by both GNU and BSD tar (2). Only xattrs `user.mime_type` and `user.meta.*` are processed. Other attributes are ignored.

In addition to the extended attributes, the object metadata and the x-delete-at/x-delete-after headers set in the request are also assigned to the extracted objects.

Notes:

(1) The POSIX 1003.1-2001 (pax) format. The default format on GNU tar 1.27.1 or later.

(2) Even with pax-format tarballs, different encoders store xattrs slightly differently; for example, GNU tar stores the xattr user.userattribute as pax header SCHILY.xattr.user.userattribute, while BSD tar (which uses libarchive) stores it as LIBARCHIVE.xattr.user.userattribute.

Response

The response from bulk operations functions differently from other Swift responses. This is because a short request body sent from the client could result in many operations on the proxy server and precautions need to be made to prevent the request from timing out due to lack of activity. To this end, the client will always receive a 200 OK response, regardless of the actual success of the call. The body of the response must be parsed to determine the actual success of the operation. In addition to this the client may receive zero or more whitespace characters prepended to the actual response body while the proxy server is completing the request.

The format of the response body defaults to text/plain but can be either json or xml depending on the `Accept` header. Acceptable formats are `text/plain`, `application/json`, `application/xml`, and `text/xml`. An example body is as follows:

```
{"Response Status": "201 Created",
 "Response Body": "",
 "Errors": [],
 "Number Files Created": 10}
```

If all valid files were uploaded successfully the Response Status will be 201 Created. If any files failed to be created the response code corresponds to the subrequests error. Possible codes are 400, 401, 502 (on server errors), etc. In both cases the response body will specify the number of files successfully uploaded and a list of the files that failed.

There are proxy logs created for each file (which becomes a subrequest) in the tar. The subrequests proxy log will have a swift.source set to EA the logs content length will reflect the unzipped size of the file. If double proxy-logging is used the leftmost logger will not have a swift.source set and the content length will reflect the size of the payload sent to the proxy (the unexpanded size of the tar.gz).

Bulk Delete

Will delete multiple objects or containers from their account with a single request. Responds to POST requests with query parameter `?bulk-delete` set. The request url is your storage url. The Content-Type should be set to `text/plain`. The body of the POST request will be a newline separated list of url encoded objects to delete. You can delete 10,000 (configurable) objects per request. The objects specified in the POST request body must be URL encoded and in the form:

```
/container_name/obj_name
```

or for a container (which must be empty at time of delete):

```
/container_name
```

The response is similar to extract archive as in every response will be a 200 OK and you must parse the response body for actual results. An example response is:

```
{"Number Not Found": 0,
 "Response Status": "200 OK",
 "Response Body": "",
 "Errors": [],
 "Number Deleted": 6}
```

If all items were successfully deleted (or did not exist), the Response Status will be 200 OK. If any failed to delete, the response code corresponds to the subrequests error. Possible codes are 400, 401, 502 (on server errors), etc. In all cases the response body will specify the number of items successfully deleted, not found, and a list of those that failed. The return body will be formatted in the way specified in the requests `Accept` header. Acceptable formats are `text/plain`, `application/json`, `application/xml`, and `text/xml`.

There are proxy logs created for each object or container (which becomes a subrequest) that is deleted. The subrequests proxy log will have a swift.source set to BD the logs content length of 0. If double proxy-logging is used the leftmost logger will not have a swift.source set and the content length will reflect the size of the payload sent to the proxy (the list of objects/containers to be deleted).

exception swift.common.middleware.bulk.**CreateContainerError**(*msg*, *status_int*, *status*)

Bases: Exception

swift.common.middleware.bulk.**get_response_body**(*data_format*, *data_dict*, *error_list*, *root_tag*)

Returns a properly formatted response body according to format.

Handles json and xml, otherwise will return text/plain. Note: xml response does not include xml declaration.

Params data_format resulting format

Params data_dict generated data about results.

Params error_list list of quoted filenames that failed

Params root_tag the tag name to use for root elements when returning XML; e.g. extract or delete

9.8.4 CatchErrors

exception swift.common.middleware.catch_errors.**BadResponseLength**

Bases: Exception

class swift.common.middleware.catch_errors.**CatchErrorMiddleware**(*app*, *conf*)

Bases: object

Middleware that provides high-level error handling and ensures that a transaction id will be set for every request.

class swift.common.middleware.catch_errors.**CatchErrorsContext**(*app*, *logger*, *trans_id_suffix=''*)

Bases: *swift.common.wsgi.WSGIContext*

swift.common.middleware.catch_errors.**enforce_byte_count**(*inner_iter*, *nbytes*)

Enforces that inner_iter yields exactly <nbytes> bytes before exhaustion.

If inner_iter fails to do so, BadResponseLength is raised.

Parameters

- **inner_iter** iterable of bytestrings
- **nbytes** number of bytes expected

9.8.5 CNAME Lookup

CNAME Lookup Middleware

Middleware that translates an unknown domain in the host header to something that ends with the configured storage_domain by looking up the given domains CNAME record in DNS.

This middleware will continue to follow a CNAME chain in DNS until it finds a record ending in the configured storage domain or it reaches the configured maximum lookup depth. If a match is found, the environments Host header is rewritten and the request is passed further down the WSGI chain.

`class swift.common.middleware.cname_lookup.CNAMELookupMiddleware(app, conf)`

Bases: `object`

CNAME Lookup Middleware

See above for a full description.

Parameters

- **app** The next WSGI filter or app in the paste.deploy chain.
- **conf** The configuration dict for the middleware.

`swift.common.middleware.cname_lookup.lookup_cname(domain, resolver)`

Given a domain, returns its DNS CNAME mapping and DNS ttl.

Parameters

- **domain** domain to query on
- **resolver** dns.resolver.Resolver() instance used for executing DNS queries

Returns (ttl, result)

9.8.6 Container Quotas

The `container_quotas` middleware implements simple quotas that can be imposed on swift containers by a user with the ability to set container metadata, most likely the account administrator. This can be useful for limiting the scope of containers that are delegated to non-admin users, exposed to `formpost` uploads, or just as a self-imposed sanity check.

Any object PUT operations that exceed these quotas return a 413 response (request entity too large) with a descriptive body.

Quotas are subject to several limitations: eventual consistency, the timeliness of the cached container_info (60 second ttl by default), and its unable to reject chunked transfer uploads that exceed the quota (though once the quota is exceeded, new chunked transfers will be refused).

Quotas are set by adding meta values to the container, and are validated when set:

Metadata	Use
X-Container-Meta-Quota-Bytes	Maximum size of the container, in bytes.
X-Container-Meta-Quota-Count	Maximum object count of the container.

The `container_quotas` middleware should be added to the pipeline in your `/etc/swift/proxy-server.conf` file just after any auth middleware. For example:

```
[pipeline:main]
pipeline = catch_errors cache tempauth container_quotas proxy-server

[filter:container_quotas]
use = egg:swift#container_quotas
```

9.8.7 Container Sync Middleware

class swift.common.middleware.container_sync.**ContainerSync**(*app*, *conf*, *logger=None*)

Bases: object

WSGI middleware that validates an incoming container sync request using the container-sync-realms.conf style of container sync.

9.8.8 Cross Domain Policies

class swift.common.middleware.crossdomain.**CrossDomainMiddleware**(*app*, *conf*, **args*, ***kwargs*)

Bases: object

Cross domain middleware used to respond to requests for cross domain policy information.

If the path is /crossdomain.xml it will respond with an xml cross domain policy document. This allows web pages hosted elsewhere to use client side technologies such as Flash, Java and Silverlight to interact with the Swift API.

To enable this middleware, add it to the pipeline in your proxy-server.conf file. It should be added before any authentication (e.g., tempauth or keystone) middleware. In this example ellipsis () indicate other middleware you may have chosen to use:

```
[pipeline:main]
pipeline =  ... crossdomain ... authtoken ... proxy-server
```

And add a filter section, such as:

```
[filter:crossdomain]
use = egg:swift#crossdomain
cross_domain_policy = <allow-access-from domain="*.example.com" />
    <allow-access-from domain="www.example.com" secure="false" />
```

For continuation lines, put some whitespace before the continuation text. Ensure you put a completely blank line to terminate the cross_domain_policy value.

The cross_domain_policy name/value is optional. If omitted, the policy defaults as if you had specified:

```
cross_domain_policy = <allow-access-from domain="*" secure="false" />
```

GET(*req*)

Returns a 200 response with cross domain policy information

9.8.9 Discoverability

Swift will by default provide clients with an interface providing details about the installation. Unless disabled (i.e `expose_info=false` in *Proxy Server Configuration*), a GET request to `/info` will return configuration data in JSON format. An example response:

```
{"swift": {"version": "1.11.0"}, "staticweb": {}, "tempurl": {}}
```

This would signify to the client that swift version 1.11.0 is running and that staticweb and tempurl are available in this installation.

There may be administrator-only information available via `/info`. To retrieve it, one must use an HMAC-signed request, similar to TempURL. The signature may be produced like so:

```
swift tempurl GET 3600 /info secret 2>/dev/null | sed s/temp_url/swiftinfo/g
```

9.8.10 Domain Remap

Domain Remap Middleware

Middleware that translates container and account parts of a domain to path parameters that the proxy server understands.

Translation is only performed when the request URLs host domain matches one of a list of domains. This list may be configured by the option `storage_domain`, and defaults to the single domain `example.com`.

If not already present, a configurable `path_root`, which defaults to `v1`, will be added to the start of the translated path.

For example, with the default configuration:

```
container.AUTH-account.example.com/object
container.AUTH-account.example.com/v1/object
```

would both be translated to:

```
container.AUTH-account.example.com/v1/AUTH_account/container/object
```

and:

```
AUTH-account.example.com/container/object
AUTH-account.example.com/v1/container/object
```

would both be translated to:

```
AUTH-account.example.com/v1/AUTH_account/container/object
```

Additionally, translation is only performed when the account name in the translated path starts with a reseller prefix matching one of a list configured by the option `reseller_prefixes`, or when no match is found but a `default_reseller_prefix` has been configured.

The `reseller_prefixes` list defaults to the single prefix `AUTH`. The `default_reseller_prefix` is not configured by default.

Browsers can convert a host header to lowercase, so the middleware checks that the reseller prefix on the account name is the correct case. This is done by comparing the items in the `reseller_prefixes` config option to the found prefix. If they match except for case, the item from `reseller_prefixes` will be used instead of the found reseller prefix. The middleware will also replace any hyphen (-) in the account name with an underscore (_).

For example, with the default configuration:

```
auth-account.example.com/container/object
AUTH-account.example.com/container/object
auth_account.example.com/container/object
AUTH_account.example.com/container/object
```

would all be translated to:

```
<unchanged>.example.com/v1/AUTH_account/container/object
```

When no match is found in `reseller_prefixes`, the `default_reseller_prefix` config option is used. When no `default_reseller_prefix` is configured, any request with an account prefix not in the `reseller_prefixes` list will be ignored by this middleware.

For example, with `default_reseller_prefix = AUTH`:

```
account.example.com/container/object
```

would be translated to:

```
account.example.com/v1/AUTH_account/container/object
```

Note that this middleware requires that container names and account names (except as described above) must be DNS-compatible. This means that the account name created in the system and the containers created by users cannot exceed 63 characters or have UTF-8 characters. These are restrictions over and above what Swift requires and are not explicitly checked. Simply put, this middleware will do a best-effort attempt to derive account and container names from elements in the domain name and put those derived values into the URL path (leaving the `Host` header unchanged).

Also note that using *Container to Container Synchronization* with remapped domain names is not advised. With *Container to Container Synchronization*, you should use the true storage end points as sync destinations.

class swift.common.middleware.domain_remap.**DomainRemapMiddleware**(*app*, *conf*)

Bases: `object`

Domain Remap Middleware

See above for a full description.

Parameters

- **app** The next WSGI filter or app in the paste.deploy chain.
- **conf** The configuration dict for the middleware.

9.8.11 Dynamic Large Objects

DLO support centers around a user specified filter that matches segments and concatenates them together in object listing order. Please see the DLO docs for *Dynamic Large Objects* further details.

9.8.12 Encryption

Encryption middleware should be deployed in conjunction with the *Keymaster* middleware.

Implements middleware for object encryption which comprises an instance of a `Decrypter` combined with an instance of an `Encrypter`.

`swift.common.middleware.crypto.`**`filter_factory`**`(`*global_conf*, ***local_conf*`)`

Provides a factory function for loading encryption middleware.

class `swift.common.middleware.crypto.encrypter.`**`EncInputWrapper`**`(`*crypto*, *keys*, *req*, *logger*`)`

Bases: `object`

File-like object to be swapped in for wsgi.input.

class `swift.common.middleware.crypto.encrypter.`**`Encrypter`**`(`*app*, *conf*`)`

Bases: `object`

Middleware for encrypting data and user metadata.

By default all PUT or POSTed object data and/or metadata will be encrypted. Encryption of new data and/or metadata may be disabled by setting the `disable_encryption` option to True. However, this middleware should remain in the pipeline in order for existing encrypted data to be read.

class `swift.common.middleware.crypto.encrypter.`**`EncrypterObjContext`**`(`*encrypter*, *logger*`)`

Bases: `swift.common.middleware.crypto.crypto_utils.CryptoWSGIContext`

`encrypt_user_metadata``(`*req*, *keys*`)`

Encrypt user-metadata header values. Replace each x-object-meta-<key> user metadata header with a corresponding x-object-transient-sysmeta-crypto-meta-<key> header which has the crypto metadata required to decrypt appended to the encrypted value.

Parameters

- **`req`** a swob Request
- **`keys`** a dict of encryption keys

`handle_post``(`*req*, *start_response*`)`

Encrypt the new object headers with a new iv and the current crypto. Note that an object may have encrypted headers while the body may remain unencrypted.

`swift.common.middleware.crypto.encrypter.`**`encrypt_header_val`**`(`*crypto*, *value*, *key*`)`

Encrypt a header value using the supplied key.

Parameters

- **`crypto`** a Crypto instance
- **`value`** value to encrypt

- **key** crypto key to use

Returns a tuple of (encrypted value, crypto_meta) where crypto_meta is a dict of form returned by `get_crypto_meta()`

Raises `ValueError` if value is empty

class swift.common.middleware.crypto.decrypter.**BaseDecrypterContext**(*crypto_app*, *server_type*, *logger*)

Bases: swift.common.middleware.crypto.crypto_utils.CryptoWSGIContext

decrypt_value(*value*, *key*, *crypto_meta*, *decoder*)

Base64-decode and decrypt a value using the crypto_meta provided.

Parameters

- **value** a base64-encoded value to decrypt
- **key** crypto key to use
- **crypto_meta** a crypto-meta dict of form returned by `get_crypto_meta()`
- **decoder** function to turn the decrypted bytes into useful data

Returns decrypted value

decrypt_value_with_meta(*value*, *key*, *required*, *decoder*)

Base64-decode and decrypt a value if crypto meta can be extracted from the value itself, otherwise return the value unmodified.

A value should either be a string that does not contain the ; character or should be of the form:

```
<base64-encoded ciphertext>;swift_meta=<crypto meta>
```

Parameters

- **value** value to decrypt
- **key** crypto key to use
- **required** if True then the value is required to be decrypted and an EncryptionException will be raised if the header cannot be decrypted due to missing crypto meta.
- **decoder** function to turn the decrypted bytes into useful data

Returns decrypted value if crypto meta is found, otherwise the unmodified value

Raises *EncryptionException* if an error occurs while parsing crypto meta or if the header value was required to be decrypted but crypto meta was not found.

get_crypto_meta(*header_name*, *check=True*)

Extract a crypto_meta dict from a header.

Parameters

- **header_name** name of header that may have crypto_meta
- **check** if True validate the crypto meta

Returns A dict containing crypto_meta items

Raises *EncryptionException* if an error occurs while parsing the crypto meta

get_decryption_keys(*req*, *crypto_meta=None*)

Determine if a response should be decrypted, and if so then fetch keys.

Parameters

- **req** a Request object
- **crypto_meta** a dict of crypto metadata

Returns a dict of decryption keys

get_unwrapped_key(*crypto_meta*, *wrapping_key*)

Get a wrapped key from crypto-meta and unwrap it using the provided wrapping key.

Parameters

- **crypto_meta** a dict of crypto-meta
- **wrapping_key** key to be used to decrypt the wrapped key

Returns an unwrapped key

Raises **HTTPInternalServerError** if the crypto-meta has no wrapped key or the unwrapped key is invalid

class swift.common.middleware.crypto.decrypter.**Decrypter**(*app*, *conf*)

Bases: object

Middleware for decrypting data and user metadata.

class swift.common.middleware.crypto.decrypter.**DecrypterContContext**(*decrypter*, *logger*)

Bases: *swift.common.middleware.crypto.decrypter.BaseDecrypterContext*

process_json_resp(*req*, *resp_iter*)

Parses json body listing and decrypt encrypted entries. Updates Content-Length header with new body length and return a body iter.

class swift.common.middleware.crypto.decrypter.**DecrypterObjContext**(*decrypter*, *logger*)

Bases: *swift.common.middleware.crypto.decrypter.BaseDecrypterContext*

decrypt_resp_headers(*put_keys*, *post_keys*)

Find encrypted headers and replace with the decrypted versions.

Parameters

- **put_keys** a dict of decryption keys used for object PUT.
- **post_keys** a dict of decryption keys used for object POST.

Returns A list of headers with any encrypted headers replaced by their decrypted values.

Raises **HTTPInternalServerError** if any error occurs while decrypting headers

multipart_response_iter(*resp*, *boundary*, *body_key*, *crypto_meta*)

Decrypts a multipart mime doc response body.

Parameters

- **resp** application response
- **boundary** multipart boundary string
- **body_key** decryption key for the response body
- **crypto_meta** crypto_meta for the response body

Returns generator for decrypted response body

response_iter(*resp*, *body_key*, *crypto_meta*, *offset*)

Decrypts a response body.

Parameters

- **resp** application response
- **body_key** decryption key for the response body
- **crypto_meta** crypto_meta for the response body
- **offset** offset into object content at which response body starts

Returns generator for decrypted response body

9.8.13 Etag Quoter

This middleware fix the Etag header of responses so that it is RFC compliant. RFC 7232 specifies that the value of the Etag header must be double quoted.

It must be placed at the beggining of the pipeline, right after cache:

```
[pipeline:main]
pipeline = ...  cache etag-quoter ...

[filter:etag-quoter]
use = egg:swift#etag_quoter
```

Set `X-Account-Rfc-Compliant-Etags: true` at the account level to have any Etags in object responses be double quoted, as in `"d41d8cd98f00b204e9800998ecf8427e"`. Alternatively, you may only fix Etags in a single container by setting `X-Container-Rfc-Compliant-Etags: true` on the container. This may be necessary for Swift to work properly with some CDNs.

Either option may also be explicitly *disabled*, so you may enable quoted Etags account-wide as above but turn them off for individual containers with `X-Container-Rfc-Compliant-Etags: false`. This may be useful if some subset of applications expect Etags to be bare MD5s.

9.8.14 FormPost

FormPost Middleware

Translates a browser form post into a regular Swift object PUT.

The format of the form is:

```
<form action="<swift-url>" method="POST"
      enctype="multipart/form-data">
  <input type="hidden" name="redirect" value="<redirect-url>" />
  <input type="hidden" name="max_file_size" value="<bytes>" />
  <input type="hidden" name="max_file_count" value="<count>" />
  <input type="hidden" name="expires" value="<unix-timestamp>" />
  <input type="hidden" name="signature" value="<hmac>" />
  <input type="file" name="file1" /><br />
  <input type="submit" />
</form>
```

Optionally, if you want the uploaded files to be temporary you can set x-delete-at or x-delete-after attributes by adding one of these as a form input:

```
<input type="hidden" name="x_delete_at" value="<unix-timestamp>" />
<input type="hidden" name="x_delete_after" value="<seconds>" />
```

If you want to specify the content type or content encoding of the files you can set content-encoding or content-type by adding them to the form input:

```
<input type="hidden" name="content-type" value="text/html" />
<input type="hidden" name="content-encoding" value="gzip" />
```

The above example applies these parameters to all uploaded files. You can also set the content-type and content-encoding on a per-file basis by adding the parameters to each part of the upload.

The <swift-url> is the URL of the Swift destination, such as:

```
https://swift-cluster.example.com/v1/AUTH_account/container/object_prefix
```

The name of each file uploaded will be appended to the <swift-url> given. So, you can upload directly to the root of container with a url like:

```
https://swift-cluster.example.com/v1/AUTH_account/container/
```

Optionally, you can include an object prefix to better separate different users uploads, such as:

```
https://swift-cluster.example.com/v1/AUTH_account/container/object_prefix
```

Note the form method must be POST and the enctype must be set as multipart/form-data.

The redirect attribute is the URL to redirect the browser to after the upload completes. This is an optional parameter. If you are uploading the form via an XMLHttpRequest the redirect should not be included. The URL will have status and message query parameters added to it, indicating the HTTP status code for the upload (2xx is success) and a possible message for further information if there was an error (such as max_file_size exceeded).

The max_file_size attribute must be included and indicates the largest single file upload that can be done, in bytes.

The max_file_count attribute must be included and indicates the maximum number of files that can be uploaded with the form. Include additional <input type="file" name="filexx" /> attributes if desired.

The expires attribute is the Unix timestamp before which the form must be submitted before it is invalidated.

The signature attribute is the HMAC signature of the form. Here is sample code for computing the signature:

```
import hmac
from hashlib import sha512
from time import time
path = '/v1/account/container/object_prefix'
redirect = 'https://srv.com/some-page'  # set to '' if redirect not in form
max_file_size = 104857600
max_file_count = 10
expires = int(time() + 600)
key = 'mykey'
hmac_body = '%s\n%s\n%s\n%s\n%s' % (path, redirect,
    max_file_size, max_file_count, expires)
signature = hmac.new(key, hmac_body, sha512).hexdigest()
```

The key is the value of either the account (X-Account-Meta-Temp-URL-Key, X-Account-Meta-Temp-Url-Key-2) or the container (X-Container-Meta-Temp-URL-Key, X-Container-Meta-Temp-Url-Key-2) TempURL keys.

Be certain to use the full path, from the /v1/ onward. Note that x_delete_at and x_delete_after are not used in signature generation as they are both optional attributes.

The command line tool `swift-form-signature` may be used (mostly just when testing) to compute expires and signature.

Also note that the file attributes must be after the other attributes in order to be processed correctly. If attributes come after the file, they wont be sent with the subrequest (there is no way to parse all the attributes on the server-side without reading the whole thing into memory to service many requests, some with large files, there just isnt enough memory on the server, so attributes following the file are simply ignored).

class swift.common.middleware.formpost.**FormPost**(*app*, *conf*, *logger=None*)

Bases: object

FormPost Middleware

See above for a full description.

The proxy logs created for any subrequests made will have swift.source set to FP.

Parameters

- **app** The next WSGI filter or app in the paste.deploy chain.
- **conf** The configuration dict for the middleware.

app

The next WSGI application/filter in the paste.deploy pipeline.

conf

The filter configuration dict.

swift.common.middleware.formpost.**MAX_VALUE_LENGTH = 4096**

The maximum size of any attributes value. Any additional data will be truncated.

swift.common.middleware.formpost.**READ_CHUNK_SIZE = 4096**

The size of data to read from the form at any given time.

swift.common.middleware.formpost.**filter_factory**(*global_conf*, ***local_conf*)

Returns the WSGI filter for use with paste.deploy.

9.8.15 GateKeeper

The `gatekeeper` middleware imposes restrictions on the headers that may be included with requests and responses. Request headers are filtered to remove headers that should never be generated by a client. Similarly, response headers are filtered to remove private headers that should never be passed to a client.

The `gatekeeper` middleware must always be present in the proxy server wsgi pipeline. It should be configured close to the start of the pipeline specified in `/etc/swift/proxy-server.conf`, immediately after catch_errors and before any other middleware. It is essential that it is configured ahead of all middlewares using system metadata in order that they function correctly.

If `gatekeeper` middleware is not configured in the pipeline then it will be automatically inserted close to the start of the pipeline by the proxy server.

swift.common.middleware.gatekeeper.**outbound_exclusions = ['x-account-sysmeta-', 'x-container-sysmeta-', 'x-object-sysmeta-', 'x-object-transient-sysmeta-', 'x-backend']**

A list of python regular expressions that will be used to match against outbound response headers. Matching headers will be removed from the response.

9.8.16 Healthcheck

class swift.common.middleware.healthcheck.**HealthCheckMiddleware**(*app*, *conf*)

Bases: `object`

Healthcheck middleware used for monitoring.

If the path is /healthcheck, it will respond 200 with OK as the body.

If the optional config parameter disable_path is set, and a file is present at that path, it will respond 503 with DISABLED BY FILE as the body.

DISABLED(*req*)

Returns a 503 response with DISABLED BY FILE in the body.

GET(*req*)

Returns a 200 response with OK in the body.

9.8.17 Keymaster

Keymaster middleware should be deployed in conjunction with the *Encryption* middleware.

class swift.common.middleware.crypto.keymaster.**BaseKeyMaster**(*app*, *conf*)

Bases: object

Base middleware for providing encryption keys.

This provides some basic helpers for:

- loading from a separate config path,
- deriving keys based on path, and
- installing a swift.callback.fetch_crypto_keys hook in the request environment.

Subclasses should define log_route, keymaster_opts, and keymaster_conf_section attributes, and implement the _get_root_secret function.

create_key(*path*, *secret_id=None*)

Creates an encryption key that is unique for the given path.

Parameters

- **path** the (WSGI string) path of the resource being encrypted.
- **secret_id** the id of the root secret from which the key should be derived.

Returns an encryption key.

Raises *UnknownSecretIdError* if the secret_id is not recognised.

class swift.common.middleware.crypto.keymaster.**KeyMaster**(*app*, *conf*)

Bases: *swift.common.middleware.crypto.keymaster.BaseKeyMaster*

Middleware for providing encryption keys.

The middleware requires its encryption root secret to be set. This is the root secret from which encryption keys are derived. This must be set before first use to a value that is at least 256 bits. The security of all encrypted data critically depends on this key, therefore it should be set to a high-entropy value. For example, a suitable value may be obtained by generating a 32 byte (or longer) value using a cryptographically secure random number generator. Changing the root secret is likely to result in data loss.

class swift.common.middleware.crypto.keymaster.**KeyMasterContext**(*keymaster*, *account*, *container*, *obj*, *meta_version_to_write='2'*)

Bases: *swift.common.wsgi.WSGIContext*

The simple scheme for key derivation is as follows: every path is associated with a key, where the key is derived from the path itself in a deterministic fashion such that the key does not need to be stored. Specifically, the key for any path is an HMAC of a root key and the path itself, calculated using an SHA256 hash function:

```
<path_key> = HMAC_SHA256(<root_secret>, <path>)
```

fetch_crypto_keys(*key_id=None*, **args*, ***kwargs*)

Setup container and object keys based on the request path.

Keys are derived from request path. The id entry in the results dict includes the part of the path used to derive keys. Other keymaster implementations may use a different strategy to generate keys and may include a different type of id, so callers should treat the id as opaque keymaster-specific data.

Parameters **key_id** if given this should be a dict with the items included under the `id` key of a dict returned by this method.

Returns A dict containing encryption keys for object and container, and entries id and all_ids. The all_ids entry is a list of key id dicts for all root secret ids including the one used to generate the returned keys.

9.8.18 KeystoneAuth

class swift.common.middleware.keystoneauth.**KeystoneAuth**(*app*, *conf*)

Bases: `object`

Swift middleware to Keystone authorization system.

In Swifts proxy-server.conf add this keystoneauth middleware and the authtoken middleware to your pipeline. Make sure you have the authtoken middleware before the keystoneauth middleware.

The authtoken middleware will take care of validating the user and keystoneauth will authorize access.

The sample proxy-server.conf shows a sample pipeline that uses keystone.

`proxy-server.conf-sample`

The authtoken middleware is shipped with keystonemiddleware - it does not have any other dependencies than itself so you can either install it by copying the file directly in your python path or by installing keystonemiddleware.

If support is required for unvalidated users (as with anonymous access) or for formpost/staticweb/tempurl middleware, authtoken will need to be configured with `delay_auth_decision` set to true. See the Keystone documentation for more detail on how to configure the authtoken middleware.

In proxy-server.conf you will need to have the setting account auto creation to true:

```
[app:proxy-server]
account_autocreate = true
```

And add a swift authorization filter section, such as:

```
[filter:keystoneauth]
use = egg:swift#keystoneauth
operator_roles = admin, swiftoperator
```

The user who is able to give ACL / create Containers permissions will be the user with a role listed in the `operator_roles` setting which by default includes the admin and the swiftoperator roles.

The keystoneauth middleware maps a Keystone project/tenant to an account in Swift by adding a prefix (`AUTH_` by default) to the tenant/project id.. For example, if the project id is `1234`, the path is `/v1/AUTH_1234`.

If you need to have a different reseller_prefix to be able to mix different auth servers you can configure the option `reseller_prefix` in your keystoneauth entry like this:

```
reseller_prefix = NEWAUTH
```

Dont forget to also update the Keystone service endpoint configuration to use NEWAUTH in the path.

It is possible to have several accounts associated with the same project. This is done by listing several prefixes as shown in the following example:

```
reseller_prefix = AUTH, SERVICE
```

This means that for project id 1234, the paths /v1/AUTH_1234 and /v1/SERVICE_1234 are associated with the project and are authorized using roles that a user has with that project. The core use of this feature is that it is possible to provide different rules for each account prefix. The following parameters may be prefixed with the appropriate prefix:

```
operator_roles
service_roles
```

For backward compatibility, if either of these parameters is specified without a prefix then it applies to all reseller_prefixes. Here is an example, using two prefixes:

```
reseller_prefix = AUTH, SERVICE
# The next three lines have identical effects (since the first applies
# to both prefixes).
operator_roles = admin, swiftoperator
AUTH_operator_roles = admin, swiftoperator
SERVICE_operator_roles = admin, swiftoperator
# The next line only applies to accounts with the SERVICE prefix
SERVICE_operator_roles = admin, some_other_role
```

X-Service-Token tokens are supported by the inclusion of the service_roles configuration option. When present, this option requires that the X-Service-Token header supply a token from a user who has a role listed in service_roles. Here is an example configuration:

```
reseller_prefix = AUTH, SERVICE
AUTH_operator_roles = admin, swiftoperator
SERVICE_operator_roles = admin, swiftoperator
SERVICE_service_roles = service
```

The keystoneauth middleware supports cross-tenant access control using the syntax `<tenant>:<user>` to specify a grantee in container Access Control Lists (ACLs). For a request to be granted by an ACL, the grantee `<tenant>` must match the UUID of the tenant to which the request X-Auth-Token is scoped and the grantee `<user>` must match the UUID of the user authenticated by that token.

Note that names must no longer be used in cross-tenant ACLs because with the introduction of domains in keystone names are no longer globally unique.

For backwards compatibility, ACLs using names will be granted by keystoneauth when it can be established that the grantee tenant, the grantee user and the tenant being accessed are either not yet in a domain (e.g. the X-Auth-Token has been obtained via the keystone v2 API) or are all in the default domain to which legacy accounts would have been migrated. The default domain is identified by its UUID, which by default has the value `default`. This can be changed by setting the `default_domain_id` option in the keystoneauth configuration:

```
default_domain_id = default
```

The backwards compatible behavior can be disabled by setting the config option `allow_names_in_acls` to false:

```
allow_names_in_acls = false
```

To enable this backwards compatibility, keystoneauth will attempt to determine the domain id of a tenant when any new account is created, and persist this as account metadata. If an account is created for a tenant using a token with reselleradmin role that is not scoped on that tenant, keystoneauth is unable to determine the domain id of the tenant; keystoneauth will assume that the tenant may not be in the default domain and therefore not match names in ACLs for that account.

By default, middleware higher in the WSGI pipeline may override auth processing, useful for middleware such as tempurl and formpost. If you know youre not going to use such middleware and you want a bit of extra security you can disable this behaviour by setting the `allow_overrides` option to `false`:

```
allow_overrides = false
```

Parameters

- **app** The next WSGI app in the pipeline
- **conf** The dict of configuration values

authorize_anonymous(*req*)

Authorize an anonymous request.

Returns None if authorization is granted, an error page otherwise.

denied_response(*req*)

Deny WSGI Response.

Returns a standard WSGI response callable with the status of 403 or 401 depending on whether the REMOTE_USER is set or not.

swift.common.middleware.keystoneauth.**filter_factory**(*global_conf*, ***local_conf*)

Returns a WSGI filter app for use with paste.deploy.

9.8.19 List Endpoints

List endpoints for an object, account or container.

This middleware makes it possible to integrate swift with software that relies on data locality information to avoid network overhead, such as Hadoop.

Using the original API, answers requests of the form:

```
/endpoints/{account}/{container}/{object}
/endpoints/{account}/{container}
/endpoints/{account}
/endpoints/v1/{account}/{container}/{object}
/endpoints/v1/{account}/{container}
/endpoints/v1/{account}
```

with a JSON-encoded list of endpoints of the form:

```
http://{server}:{port}/{dev}/{part}/{acc}/{cont}/{obj}
http://{server}:{port}/{dev}/{part}/{acc}/{cont}
http://{server}:{port}/{dev}/{part}/{acc}
```

correspondingly, e.g.:

```
http://10.1.1.1:6200/sda1/2/a/c2/o1
http://10.1.1.1:6200/sda1/2/a/c2
http://10.1.1.1:6200/sda1/2/a
```

Using the v2 API, answers requests of the form:

```
/endpoints/v2/{account}/{container}/{object}
/endpoints/v2/{account}/{container}
/endpoints/v2/{account}
```

with a JSON-encoded dictionary containing a key endpoints that maps to a list of endpoints having the same form as described above, and a key headers that maps to a dictionary of headers that should be sent with a request made to the endpoints, e.g.:

```
{ "endpoints": {"http://10.1.1.1:6210/sda1/2/a/c3/o1",
                "http://10.1.1.1:6230/sda3/2/a/c3/o1",
                "http://10.1.1.1:6240/sda4/2/a/c3/o1"},
  "headers": {"X-Backend-Storage-Policy-Index": "1"}}
```

In this example, the headers dictionary indicates that requests to the endpoint URLs should include the header X-Backend-Storage-Policy-Index: 1 because the objects container is using storage policy index 1.

The /endpoints/ path is customizable (list_endpoints_path configuration parameter).

Intended for consumption by third-party services living inside the cluster (as the endpoints make sense only inside the cluster behind the firewall); potentially written in a different language.

This is why its provided as a REST API and not just a Python API: to avoid requiring clients to write their own ring parsers in their languages, and to avoid the necessity to distribute the ring file to clients and keep it up-to-date.

Note that the call is not authenticated, which means that a proxy with this middleware enabled should not be open to an untrusted environment (everyone can query the locality data using this middleware).

class swift.common.middleware.list_endpoints.**ListEndpointsMiddleware**(*app*, *conf*)

Bases: object

List endpoints for an object, account or container.

See above for a full description.

Uses configuration parameter *swift_dir* (default */etc/swift*).

Parameters

- **app** The next WSGI filter or app in the paste.deploy chain.
- **conf** The configuration dict for the middleware.

get_object_ring(*policy_idx*)

Get the ring object to use to handle a request based on its policy.

Policy_idx policy index as defined in swift.conf

Returns appropriate ring object

9.8.20 Memcache

class swift.common.middleware.memcache.**MemcacheMiddleware**(*app*, *conf*)

Bases: object

Caching middleware that manages caching in swift.

9.8.21 Name Check (Forbidden Character Filter)

Created on February 27, 2012

A filter that disallows any paths that contain defined forbidden characters or that exceed a defined length.

Place early in the proxy-server pipeline after the left-most occurrence of the `proxy-logging` middleware (if present) and before the final `proxy-logging` middleware (if present) or the `proxy-serer` app itself, e.g.:

```
[pipeline:main]
pipeline = catch_errors healthcheck proxy-logging name_check cache ratelimit␣
↪tempauth sos proxy-logging proxy-server

[filter:name_check]
use = egg:swift#name_check
forbidden_chars = '"`<>
maximum_length = 255
```

There are default settings for forbidden_chars (FORBIDDEN_CHARS) and maximum_length (MAX_LENGTH)

The filter returns HTTPBadRequest if path is invalid.

@author: eamonn-otoole

9.8.22 Object Versioning

Object versioning in Swift has 3 different modes. There are two *legacy modes* that have similar API with a slight difference in behavior and this middleware introduces a new mode with a completely redesigned API and implementation.

In terms of the implementation, this middleware relies heavily on the use of static links to reduce the amount of backend data movement that was part of the two legacy modes. It also introduces a new API for enabling the feature and to interact with older versions of an object.

Compatibility between modes

This new mode is not backwards compatible or interchangeable with the two legacy modes. This means that existing containers that are being versioned by the two legacy modes cannot enable the new mode. The new mode can only be enabled on a new container or a container without either `X-Versions-Location` or `X-History-Location` header set. Attempting to enable the new mode on a container with either header will result in a `400 Bad Request` response.

Enable Object Versioning in a Container

After the introduction of this feature containers in a Swift cluster will be in one of either 3 possible states: 1. Object versioning never enabled, 2. Object Versioning Enabled or 3. Object Versioning Disabled. Once versioning has been enabled on a container, it will always have a flag stating whether it is either enabled or disabled.

Clients enable object versioning on a container by performing either a PUT or POST request with the header `X-Versions-Enabled: true`. Upon enabling the versioning for the first time, the middleware will create a hidden container where object versions are stored. This hidden container will inherit the same Storage Policy as its parent container.

To disable, clients send a POST request with the header `X-Versions-Enabled: false`. When versioning is disabled, the old versions remain unchanged.

To delete a versioned container, versioning must be disabled and all versions of all objects must be deleted before the container can be deleted. At such time, the hidden container will also be deleted.

Object CRUD Operations to a Versioned Container

When data is `PUT` into a versioned container (a container with the versioning flag enabled), the actual object is written to a hidden container and a symlink object is written to the parent container. Every object is assigned a version id. This id can be retrieved from the `X-Object-Version-Id` header in the PUT response.

Note: When object versioning is disabled on a container, new data will no longer be versioned, but older versions remain untouched. Any new data `PUT` will result in a object with a `null` version-id. The versioning API can be used to both list and operate on previous versions even while versioning is disabled.

If versioning is re-enabled and an overwrite occurs on a *null* id object. The object will be versioned off with a regular version-id.

A GET to a versioned object will return the current version of the object. The X-Object-Version-Id header is also returned in the response.

A POST to a versioned object will update the most current object metadata as normal, but will not create a new version of the object. In other words, new versions are only created when the content of the object changes.

On DELETE, the middleware will write a zero-byte delete marker object version that notes **when** the delete took place. The symlink object will also be deleted from the versioned container. The object will no longer appear in container listings for the versioned container and future requests there will return 404 Not Found. However, the previous versions content will still be recoverable.

Object Versioning API

Clients can now operate on previous versions of an object using this new versioning API.

First to list previous versions, issue a a GET request to the versioned container with query parameter:

```
?versions
```

To list a container with a large number of object versions, clients can also use the version_marker parameter together with the marker parameter. While the marker parameter is used to specify an object name the version_marker will be used specify the version id.

All other pagination parameters can be used in conjunction with the versions parameter.

During container listings, delete markers can be identified with the content-type application/x-deleted;swift_versions_deleted=1. The most current version of an object can be identified by the field is_latest.

To operate on previous versions, clients can use the query parameter:

```
?version-id=<id>
```

where the <id> is the value from the X-Object-Version-Id header.

Only COPY, HEAD, GET and DELETE operations can be performed on previous versions. Either a PUT or POST request with a version-id parameter will result in a 400 Bad Request response.

A HEAD/GET request to a delete-marker will result in a 404 Not Found response.

When issuing DELETE requests with a version-id parameter, delete markers are not written down. A DELETE request with a version-id parameter to the current object will result in a both the symlink and the backing data being deleted. A DELETE to any other version will result in that version only be deleted and no changes made to the symlink pointing to the current version.

How to Enable Object Versioning in a Swift Cluster

To enable this new mode in a Swift cluster the `versioned_writes` and `symlink` middlewares must be added to the proxy pipeline, you must also set the option `allow_object_versioning` to `True`.

class swift.common.middleware.versioned_writes.object_versioning.**AccountContext**(*wsgi_app*, *logger*)

Bases: *swift.common.middleware.versioned_writes.object_versioning.ObjectVersioningContext*

class swift.common.middleware.versioned_writes.object_versioning.**ByteCountingReader**(*file_like*)

Bases: `object`

Counts bytes read from file_like so we know how big the object is that the client just PUT.

This is particularly important when the client sends a chunk-encoded body, so we dont have a Content-Length header available.

class swift.common.middleware.versioned_writes.object_versioning.**ContainerContext**(*wsgi_app*, *logger*)

Bases: *swift.common.middleware.versioned_writes.object_versioning.ObjectVersioningContext*

handle_delete(*req*, *start_response*)

Handle request to delete a users container.

As part of deleting a container, this middleware will also delete the hidden container holding object versions.

Before a users container can be deleted, swift must check if there are still old object versions from that container. Only after disabling versioning and deleting *all* object versions can a container be deleted.

handle_request(*req*, *start_response*)

Handle request for container resource.

On PUT, POST set version location and enabled flag sysmeta. For container listings of a versioned container, update the objects bytes and etag to use the targets instead of using the symlink info.

class swift.common.middleware.versioned_writes.object_versioning.**ObjectContext**(*wsgi_app*, *logger*)

Bases: *swift.common.middleware.versioned_writes.object_versioning.ObjectVersioningContext*

handle_delete(*req*, *versions_cont*, *api_version*, *account_name*, *container_name*, *object_name*, *is_enabled*)

Handle DELETE requests.

Copy current version of object to versions_container and write a delete marker before proceeding with original request.

Parameters

- **req** original request.
- **versions_cont** container where previous versions of the object are stored.
- **api_version** api version.
- **account_name** account name.
- **object_name** name of object of original request

handle_post(*req*, *versions_cont*, *account*)

Handle a POST request to an object in a versioned container.

If the response is a 307 because the POST went to a symlink, follow the symlink and send the request to the versioned object

Parameters

- **req** original request.
- **versions_cont** container where previous versions of the object are stored.
- **account** account name.

handle_put(*req*, *versions_cont*, *api_version*, *account_name*, *object_name*, *is_enabled*)

Check if the current version of the object is a versions-symlink if not, its because this object was added to the container when versioning was not enabled. Well need to copy it into the versions containers now that versioning is enabled.

Also, put the new data from the client into the versions container and add a static symlink in the versioned container.

Parameters

- **req** original request.
- **versions_cont** container where previous versions of the object are stored.
- **api_version** api version.
- **account_name** account name.
- **object_name** name of object of original request

handle_put_version(*req*, *versions_cont*, *api_version*, *account_name*, *container*, *object_name*, *is_enabled*, *version*)

Handle a PUT?version-id request and create/update the is_latest link to point to the specific version. Expects a valid version id.

handle_versioned_request(*req*, *versions_cont*, *api_version*, *account*, *container*, *obj*, *is_enabled*, *version*)

Handle version-id request for object resource. When a request contains a `version-id=<id>` parameter, the request is acted upon the actual version of that object. Version-aware operations require that the container is versioned, but do not require that the versioning is currently enabled. Users should be able to operate on older versions of an object even if versioning is currently suspended.

PUT and POST requests are not allowed as that would overwrite the contents of the versioned object.

Parameters

- **req** The original request
- **versions_cont** container holding versions of the requested obj
- **api_version** should be v1 unless swift bumps api version
- **account** account name string
- **container** container name string
- **object** object name string
- **is_enabled** is versioning currently enabled
- **version** version of the object to act on

class swift.common.middleware.versioned_writes.object_versioning.**ObjectVersioningContext**(*wsg*
log
ger

Bases: *swift.common.wsgi.WSGIContext*

9.8.23 Proxy Logging

Logging middleware for the Swift proxy.

This serves as both the default logging implementation and an example of how to plug in your own logging format/method.

The logging format implemented below is as follows:

client_ip remote_addr end_time.datetime method path protocol status_int referer user_agent auth_token bytes_recvd bytes_sent client_etag transaction_id headers request_time source log_info start_time end_time policy_index

These values are space-separated, and each is url-encoded, so that they can be separated with a simple .split()

- remote_addr is the contents of the REMOTE_ADDR environment variable, while client_ip is swifts best guess at the end-user IP, extracted variously from the X-Forwarded-For header, X-Cluster-Ip header, or the REMOTE_ADDR environment variable.
- source (swift.source in the WSGI environment) indicates the code that generated the request, such as most middleware. (See below for more detail.)
- log_info (swift.log_info in the WSGI environment) is for additional information that could prove quite useful, such as any x-delete-at value or other behind the scenes activity that might not otherwise be detectable from the plain log information. Code that wishes to add additional log information should use code like `env.setdefault('swift.log_info', []).append(your_info)` so as to not disturb others log information.
- Values that are missing (e.g. due to a header not being present) or zero are generally represented by a single hyphen (-).

The proxy-logging can be used twice in the proxy servers pipeline when there is middleware installed that can return custom responses that dont follow the standard pipeline to the proxy server.

For example, with staticweb, the middleware might intercept a request to /v1/AUTH_acc/cont/, make a subrequest to the proxy to retrieve /v1/AUTH_acc/cont/index.html and, in effect, respond to the clients

original request using the 2nd requests body. In this instance the subrequest will be logged by the rightmost middleware (with a swift.source set) and the outgoing request (with body overridden) will be logged by leftmost middleware.

Requests that follow the normal pipeline (use the same wsgi environment throughout) will not be double logged because an environment variable (swift.proxy_access_log_made) is checked/set when a log is made.

All middleware making subrequests should take care to set swift.source when needed. With the doubled proxy logs, any consumer/processor of swifts proxy logs should look at the swift.source field, the rightmost log value, to decide if this is a middleware subrequest or not. A log processor calculating bandwidth usage will want to only sum up logs with no swift.source.

class swift.common.middleware.proxy_logging.**ProxyLoggingMiddleware**(*app*, *conf*, *logger=None*)

Bases: object

Middleware that logs Swift proxy requests in the swift log format.

log_request(*req*, *status_int*, *bytes_received*, *bytes_sent*, *start_time*, *end_time*, *resp_headers=None*, *ttfb=0*, *wire_status_int=None*)

Log a request.

Parameters

- **req** swob.Request object for the request
- **status_int** integer code for the response status
- **bytes_received** bytes successfully read from the request body
- **bytes_sent** bytes yielded to the WSGI server
- **start_time** timestamp request started
- **end_time** timestamp request completed
- **resp_headers** dict of the response headers
- **wire_status_int** the on the wire status int

9.8.24 Ratelimit

exception swift.common.middleware.ratelimit.**MaxSleepTimeHitError**

Bases: Exception

class swift.common.middleware.ratelimit.**RateLimitMiddleware**(*app*, *conf*, *logger=None*)

Bases: object

Rate limiting middleware

Rate limits requests on both an Account and Container level. Limits are configurable.

get_ratelimitable_key_tuples(*req*, *account_name*, *container_name=None*, *obj_name=None*, *global_ratelimit=None*)

Returns a list of key (used in memcache), ratelimit tuples. Keys should be checked in order.

Parameters

- **req** swob request
- **account_name** account name from path
- **container_name** container name from path
- **obj_name** object name from path
- **global_ratelimit** this account has an account wide ratelimit on all writes combined

handle_ratelimit(*req*, *account_name*, *container_name*, *obj_name*)

Performs rate limiting and account white/black listing. Sleeps if necessary. If self.memcache_client is not set, immediately returns None.

Parameters

- **account_name** account name from path
- **container_name** container name from path
- **obj_name** object name from path

swift.common.middleware.ratelimit.**filter_factory**(*global_conf*, ***local_conf*)

paste.deploy app factory for creating WSGI proxy apps.

swift.common.middleware.ratelimit.**get_maxrate**(*ratelimits*, *size*)

Returns number of requests allowed per second for given size.

swift.common.middleware.ratelimit.**interpret_conf_limits**(*conf*, *name_prefix*, *info=None*)

Parses general parms for rate limits looking for things that start with the provided name_prefix within the provided conf and returns lists for both internal use and for /info

Parameters

- **conf** conf dict to parse
- **name_prefix** prefix of config parms to look for
- **info** set to return extra stuff for /info registration

9.8.25 Read Only

class swift.common.middleware.read_only.**ReadOnlyMiddleware**(*app*, *conf*, *logger=None*)

Bases: object

Middleware that make an entire cluster or individual accounts read only.

account_read_only(*req*, *account*)

Check whether an account should be read-only.

This considers both the cluster-wide config value as well as the per-account override in X-Account-Sysmeta-Read-Only.

swift.common.middleware.read_only.**filter_factory**(*global_conf*, ***local_conf*)

paste.deploy app factory for creating WSGI proxy apps.

9.8.26 Recon

class swift.common.middleware.recon.**ReconMiddleware**(*app*, *conf*, **args*, ***kwargs*)

Bases: object

Recon middleware used for monitoring.

/recon/load|mem|async will return various system metrics.

Needs to be added to the pipeline and requires a filter declaration in the [account|container|object]-server conf file:

[filter:recon] use = egg:swift#recon recon_cache_path = /var/cache/swift

get_async_info()

get # of async pendings

get_auditor_info(*recon_type*)

get auditor info

get_device_info()

get devices

get_diskusage()

get disk utilization statistics

get_driveaudit_error()

get # of drive audit errors

get_expirer_info(*recon_type*)

get expirer info

get_load(*openr=<built-in function open>*)

get info from /proc/loadavg

get_mem(*openr=<built-in function open>*)

get info from /proc/meminfo

get_mounted(*openr=<built-in function open>*)

get ALL mounted fs from /proc/mounts

get_quarantine_count()

get obj/container/account quarantine counts

get_reconstruction_info()

get reconstruction info

get_relinker_info()

get relinker info, if any

get_replication_info(*recon_type*)

get replication info

get_ring_md5()

get all ring md5sums

get_sharding_info()

get sharding info

get_socket_info(*openr=<built-in function open>*)

get info from /proc/net/sockstat and sockstat6

Note: The mem value is actually kernel pages, but we return bytes allocated based on the systems page size.

get_swift_conf_md5()

get md5 of swift.conf

get_time()

get current time

get_unmounted()

list unmounted (failed?) devices

get_updater_info(*recon_type*)

get updater info

get_version()

get swift version

9.8.27 Server Side Copy

Server side copy is a feature that enables users/clients to COPY objects between accounts and containers without the need to download and then re-upload objects, thus eliminating additional bandwidth consumption and also saving time. This may be used when renaming/moving an object which in Swift is a (COPY + DELETE) operation.

The server side copy middleware should be inserted in the pipeline after auth and before the quotas and large object middlewares. If it is not present in the pipeline in the proxy-server configuration file, it will be inserted automatically. There is no configurable option provided to turn off server side copy.

Metadata

- All metadata of source object is preserved during object copy.
- One can also provide additional metadata during PUT/COPY request. This will over-write any existing conflicting keys.
- Server side copy can also be used to change content-type of an existing object.

Object Copy

- The destination container must exist before requesting copy of the object.
- When several replicas exist, the system copies from the most recent replica. That is, the copy operation behaves as though the X-Newest header is in the request.
- The request to copy an object should have no body (i.e. content-length of the request must be zero).

There are two ways in which an object can be copied:

1. Send a PUT request to the new object (destination/target) with an additional header named `X-Copy-From` specifying the source object (in /container/object format). Example:

```
curl -i -X PUT http://<storage_url>/container1/destination_obj
 -H 'X-Auth-Token: <token>'
 -H 'X-Copy-From: /container2/source_obj'
 -H 'Content-Length: 0'
```

2. Send a COPY request with an existing object in URL with an additional header named `Destination` specifying the destination/target object (in /container/object format). Example:

```
curl -i -X COPY http://<storage_url>/container2/source_obj
 -H 'X-Auth-Token: <token>'
 -H 'Destination: /container1/destination_obj'
 -H 'Content-Length: 0'
```

Note that if the incoming request has some conditional headers (e.g. `Range`, `If-Match`), the *source* object will be evaluated for these headers (i.e. if PUT with both `X-Copy-From` and `Range`, Swift will make a partial copy to the destination object).

Cross Account Object Copy

Objects can also be copied from one account to another account if the user has the necessary permissions (i.e. permission to read from container in source account and permission to write to container in destination account).

Similar to examples mentioned above, there are two ways to copy objects across accounts:

1. Like the example above, send PUT request to copy object but with an additional header named `X-Copy-From-Account` specifying the source account. Example:

```
curl -i -X PUT http://<host>:<port>/v1/AUTH_test1/container/destination_
↪obj
 -H 'X-Auth-Token: <token>'
 -H 'X-Copy-From: /container/source_obj'
 -H 'X-Copy-From-Account: AUTH_test2'
 -H 'Content-Length: 0'
```

2. Like the previous example, send a COPY request but with an additional header named `Destination-Account` specifying the name of destination account. Example:

```
curl -i -X COPY http://<host>:<port>/v1/AUTH_test2/container/source_obj
 -H 'X-Auth-Token: <token>'
```

(continues on next page)

(continued from previous page)

```
-H 'Destination: /container/destination_obj'
-H 'Destination-Account: AUTH_test1'
-H 'Content-Length: 0'
```

Large Object Copy

The best option to copy a large object is to copy segments individually. To copy the manifest object of a large object, add the query parameter to the copy request:

```
?multipart-manifest=get
```

If a request is sent without the query parameter, an attempt will be made to copy the whole object but will fail if the object size is greater than 5GB.

class swift.common.middleware.copy.**ServerSideCopyWebContext**(*app*, *logger*)

Bases: *swift.common.wsgi.WSGIContext*

9.8.28 Static Large Objects

Please see the SLO docs for *Static Large Objects* further details.

9.8.29 StaticWeb

This StaticWeb WSGI middleware will serve container data as a static web site with index file and error file resolution and optional file listings. This mode is normally only active for anonymous requests. When using keystone for authentication set `delay_auth_decision = true` in the authtoken middleware configuration in your `/etc/swift/proxy-server.conf` file. If you want to use it with authenticated requests, set the `X-Web-Mode: true` header on the request.

The `staticweb` filter should be added to the pipeline in your `/etc/swift/proxy-server.conf` file just after any auth middleware. Also, the configuration section for the `staticweb` middleware itself needs to be added. For example:

```
[DEFAULT]
...

[pipeline:main]
pipeline = catch_errors healthcheck proxy-logging cache ratelimit tempauth
           staticweb proxy-logging proxy-server


...

[filter:staticweb]
use = egg:swift#staticweb
```

Any publicly readable containers (for example, `X-Container-Read: .r:*`, see *ACLs* for more information on this) will be checked for X-Container-Meta-Web-Index and X-Container-Meta-Web-Error header values:

```
X-Container-Meta-Web-Index  <index.name>
X-Container-Meta-Web-Error  <error.name.suffix>
```

If X-Container-Meta-Web-Index is set, any <index.name> files will be served without having to specify the <index.name> part. For instance, setting `X-Container-Meta-Web-Index: index.html` will be able to serve the object /pseudo/path/index.html with just /pseudo/path or /pseudo/path/

If X-Container-Meta-Web-Error is set, any errors (currently just 401 Unauthorized and 404 Not Found) will instead serve the /<status.code><error.name.suffix> object. For instance, setting `X-Container-Meta-Web-Error: error.html` will serve /404error.html for requests for paths not found.

For pseudo paths that have no <index.name>, this middleware can serve HTML file listings if you set the `X-Container-Meta-Web-Listings: true` metadata item on the container.

If listings are enabled, the listings can have a custom style sheet by setting the X-Container-Meta-Web-Listings-CSS header. For instance, setting `X-Container-Meta-Web-Listings-CSS: listing.css` will make listings link to the /listing.css style sheet. If you view source in your browser on a listing page, you will see the well defined document structure that can be styled.

By default, the listings will be rendered with a label of Listing of /v1/account/container/path. This can be altered by setting a `X-Container-Meta-Web-Listings-Label: <label>`. For example, if the label is set to example.com, a label of Listing of example.com/path will be used instead.

The content-type of directory marker objects can be modified by setting the `X-Container-Meta-Web-Directory-Type` header. If the header is not set, application/directory is used by default. Directory marker objects are 0-byte objects that represent directories to create a simulated hierarchical structure.

Example usage of this middleware via `swift`:

Make the container publicly readable:

```
swift post -r '.r:*' container
```

You should be able to get objects directly, but no index.html resolution or listings.

Set an index file directive:

```
swift post -m 'web-index:index.html' container
```

You should be able to hit paths that have an index.html without needing to type the index.html part.

Turn on listings:

```
swift post -r '.r:*,.rlistings' container
swift post -m 'web-listings: true' container
```

Now you should see object listings for paths and pseudo paths that have no index.html.

Enable a custom listings style sheet:

```
swift post -m 'web-listings-css:listings.css' container
```

Set an error file:

```
swift post -m 'web-error:error.html' container
```

Now 401s should load 401error.html, 404s should load 404error.html, etc.

Set Content-Type of directory marker object:

```
swift post -m 'web-directory-type:text/directory' container
```

Now 0-byte objects with a content-type of text/directory will be treated as directories rather than objects.

class swift.common.middleware.staticweb.**StaticWeb**(*app*, *conf*)

Bases: object

The Static Web WSGI middleware filter; serves container data as a static web site. See *staticweb* for an overview.

The proxy logs created for any subrequests made will have swift.source set to SW.

Parameters

- **app** The next WSGI application/filter in the paste.deploy pipeline.
- **conf** The filter configuration dict.

app

The next WSGI application/filter in the paste.deploy pipeline.

conf

The filter configuration dict. Only used in tests.

swift.common.middleware.staticweb.**filter_factory**(*global_conf*, ***local_conf*)

Returns a Static Web WSGI filter for use with paste.deploy.

9.8.30 Symlink

Symlink Middleware

Symlinks are objects stored in Swift that contain a reference to another object (hereinafter, this is called target object). They are analogous to symbolic links in Unix-like operating systems. The existence of a symlink object does not affect the target object in any way. An important use case is to use a path in one container to access an object in a different container, with a different policy. This allows policy cost/performance trade-offs to be made on individual objects.

Clients create a Swift symlink by performing a zero-length PUT request with the header `X-Symlink-Target: <container>/<object>`. For a cross-account symlink, the header `X-Symlink-Target-Account: <account>` must be included. If omitted, it is inserted automatically with the account of the symlink object in the PUT request process.

Symlinks must be zero-byte objects. Attempting to PUT a symlink with a non-empty request body will result in a 400-series error. Also, POST with `X-Symlink-Target` header always results in a 400-series error. The target object need not exist at symlink creation time.

Clients may optionally include a `X-Symlink-Target-Etag: <etag>` header during the PUT. If present, this will create a static symlink instead of a dynamic symlink. Static symlinks point to a specific object rather than a specific name. They do this by using the value set in their `X-Symlink-Target-Etag`

header when created to verify it still matches the ETag of the object theyre pointing at on a GET. In contrast to a dynamic symlink the target object referenced in the `X-Symlink-Target` header must exist and its ETag must match the `X-Symlink-Target-Etag` or the symlink creation will return a client error.

A GET/HEAD request to a symlink will result in a request to the target object referenced by the symlinks `X-Symlink-Target-Account` and `X-Symlink-Target` headers. The response of the GET/HEAD request will contain a `Content-Location` header with the path location of the target object. A GET/HEAD request to a symlink with the query parameter `?symlink=get` will result in the request targeting the symlink itself.

A symlink can point to another symlink. Chained symlinks will be traversed until the target is not a symlink. If the number of chained symlinks exceeds the limit `symloop_max` an error response will be produced. The value of `symloop_max` can be defined in the symlink config section of *proxy-server.conf*. If not specified, the default `symloop_max` value is 2. If a value less than 1 is specified, the default value will be used.

If a static symlink (i.e. a symlink created with a `X-Symlink-Target-Etag` header) targets another static symlink, both of the `X-Symlink-Target-Etag` headers must match the target object for the GET to succeed. If a static symlink targets a dynamic symlink (i.e. a symlink created without a `X-Symlink-Target-Etag` header) then the `X-Symlink-Target-Etag` header of the static symlink must be the Etag of the zero-byte object. If a symlink with a `X-Symlink-Target-Etag` targets a large object manifest it must match the ETag of the manifest (e.g. the ETag as returned by `multipart-manifest=get` or value in the `X-Manifest-Etag` header).

A HEAD/GET request to a symlink object behaves as a normal HEAD/GET request to the target object. Therefore issuing a HEAD request to the symlink will return the target metadata, and issuing a GET request to the symlink will return the data and metadata of the target object. To return the symlink metadata (with its empty body) a GET/HEAD request with the `?symlink=get` query parameter must be sent to a symlink object.

A POST request to a symlink will result in a 307 Temporary Redirect response. The response will contain a `Location` header with the path of the target object as the value. The request is never redirected to the target object by Swift. Nevertheless, the metadata in the POST request will be applied to the symlink because object servers cannot know for sure if the current object is a symlink or not in eventual consistency.

A symlinks `Content-Type` is completely independent from its target. As a convenience Swift will automatically set the `Content-Type` on a symlink PUT if not explicitly set by the client. If the client sends a `X-Symlink-Target-Etag` Swift will set the symlinks `Content-Type` to that of the target, otherwise it will be set to `application/symlink`. You can review a symlinks `Content-Type` using the `?symlink=get` interface. You can change a symlinks `Content-Type` using a POST request. The symlinks `Content-Type` will appear in the container listing.

A DELETE request to a symlink will delete the symlink itself. The target object will not be deleted.

A COPY request, or a PUT request with a `X-Copy-From` header, to a symlink will copy the target object. The same request to a symlink with the query parameter `?symlink=get` will copy the symlink itself.

An OPTIONS request to a symlink will respond with the options for the symlink only; the request will not be redirected to the target object. Please note that if the symlinks target object is in another container with CORS settings, the response will not reflect the settings.

Tempurls can be used to GET/HEAD symlink objects, but PUT is not allowed and will result in a 400-series error. The GET/HEAD tempurls honor the scope of the tempurl key. Container tempurl will only work on symlinks where the target container is the same as the symlink. In case a symlink targets an object

in a different container, a GET/HEAD request will result in a 401 Unauthorized error. The account level tempurl will allow cross-container symlinks, but not cross-account symlinks.

If a symlink object is overwritten while it is in a versioned container, the symlink object itself is versioned, not the referenced object.

A GET request with query parameter `?format=json` to a container which contains symlinks will respond with additional information `symlink_path` for each symlink object in the container listing. The `symlink_path` value is the target path of the symlink. Clients can differentiate symlinks and other objects by this function. Note that responses in any other format (e.g. `?format=xml`) wont include `symlink_path` info. If a `X-Symlink-Target-Etag` header was included on the symlink, JSON container listings will include that value in a `symlink_etag` key and the target objects `Content-Length` will be included in the key `symlink_bytes`.

If a static symlink targets a static large object manifest it will carry forward the SLOs size and slo_etag in the container listing using the `symlink_bytes` and `slo_etag` keys. However, manifests created before swift v2.12.0 (released Dec 2016) do not contain enough metadata to propagate the extra SLO information to the listing. Clients may recreate the manifest (COPY w/ `?multipart-manfiest=get`) before creating a static symlink to add the requisite metadata.

Errors

- PUT with the header `X-Symlink-Target` with non-zero Content-Length will produce a 400 BadRequest error.
- POST with the header `X-Symlink-Target` will produce a 400 BadRequest error.
- GET/HEAD traversing more than `symloop_max` chained symlinks will produce a 409 Conflict error.
- PUT/GET/HEAD on a symlink that inclues a `X-Symlink-Target-Etag` header that does not match the target will poduce a 409 Conflict error.
- POSTs will produce a 307 Temporary Redirect error.

Deployment

Symlinks are enabled by adding the *symlink* middleware to the proxy server WSGI pipeline and including a corresponding filter configuration section in the *proxy-server.conf* file. The *symlink* middleware should be placed after *slo*, *dlo* and *versioned_writes* middleware, but before *encryption* middleware in the pipeline. See the *proxy-server.conf-sample* file for further details. *Additional steps* are required if the container sync feature is being used.

Note: Once you have deployed *symlink* middleware in your pipeline, you should neither remove the *symlink* middleware nor downgrade swift to a version earlier than symlinks being supported. Doing so may result in unexpected container listing results in addition to symlink objects behaving like a normal object.

Container sync configuration

If container sync is being used then the *symlink* middleware must be added to the container sync internal client pipeline. The following configuration steps are required:

1. Create a custom internal client configuration file for container sync (if one is not already in use) based on the sample file *internal-client.conf-sample*. For example, copy *internal-client.conf-sample* to */etc/swift/container-sync-client.conf*.
2. Modify this file to include the *symlink* middleware in the pipeline in the same way as described above for the proxy server.
3. Modify the container-sync section of all container server config files to point to this internal client config file using the `internal_client_conf_path` option. For example:

```
internal_client_conf_path = /etc/swift/container-sync-client.conf
```

Note: These container sync configuration steps will be necessary for container sync probe tests to pass if the *symlink* middleware is included in the proxy pipeline of a test cluster.

---

class swift.common.middleware.symlink.**SymlinkContainerContext**(*wsgi_app*, *logger*)

Bases: *swift.common.wsgi.WSGIContext*

handle_container(*req*, *start_response*)

Handle container requests.

Parameters

- **req** a *Request*
- **start_response** start_response function

Returns Response Iterator after start_response called.

class swift.common.middleware.symlink.**SymlinkMiddleware**(*app*, *conf*, *symloop_max*)

Bases: object

Middleware that implements symlinks.

Symlinks are objects stored in Swift that contain a reference to another object (i.e., the target object). An important use case is to use a path in one container to access an object in a different container, with a different policy. This allows policy cost/performance trade-offs to be made on individual objects.

class swift.common.middleware.symlink.**SymlinkObjectContext**(*wsgi_app*, *logger*, *symloop_max*)

Bases: *swift.common.wsgi.WSGIContext*

handle_get_head(*req*)

Handle get/head request and in case the response is a symlink, redirect request to target object.

Parameters **req** HTTP GET or HEAD object request

Returns Response Iterator

handle_get_head_symlink(*req*)

Handle get/head request when client sent parameter ?symlink=get

Parameters **req** HTTP GET or HEAD object request with param ?symlink=get

Returns Response Iterator

handle_object(*req*, *start_response*)

Handle object requests.

Parameters

- **req** a *Request*
- **start_response** start_response function

Returns Response Iterator after start_response has been called

handle_post(*req*)

Handle post request. If POSTing to a symlink, a HTTPTemporaryRedirect error message is returned to client.

Clients that POST to symlinks should understand that the POST is not redirected to the target object like in a HEAD/GET request. POSTs to a symlink will be handled just like a normal object by the object server. It cannot reject it because it may not have symlink state when the POST lands. The object server has no knowledge of what is a symlink object is. On the other hand, on POST requests, the object server returns all sysmeta of the object. This method uses that sysmeta to determine if the stored object is a symlink or not.

Parameters **req** HTTP POST object request

Raises HTTPTemporaryRedirect if POSTing to a symlink.

Returns Response Iterator

handle_put(*req*)

Handle put request when it contains X-Symlink-Target header.

Symlink headers are validated and moved to sysmeta namespace. :param req: HTTP PUT object request :returns: Response Iterator

swift.common.middleware.symlink.**symlink_sysmeta_to_usermeta**(*headers*)

Helper function to translate from cluster-facing X-Object-Sysmeta-Symlink-* headers to client-facing X-Symlink-* headers.

Parameters **headers** request headers dict. Note that the headers dict will be updated directly.

swift.common.middleware.symlink.**symlink_usermeta_to_sysmeta**(*headers*)

Helper function to translate from client-facing X-Symlink-* headers to cluster-facing X-Object-Sysmeta-Symlink-* headers.

Parameters **headers** request headers dict. Note that the headers dict will be updated directly.

9.8.31 TempAuth

Test authentication and authorization system.

Add to your pipeline in proxy-server.conf, such as:

```
[pipeline:main]
pipeline = catch_errors cache tempauth proxy-server
```

Set account auto creation to true in proxy-server.conf:

```
[app:proxy-server]
account_autocreate = true
```

And add a tempauth filter section, such as:

```
[filter:tempauth]
use = egg:swift#tempauth
user_admin_admin = admin .admin .reseller_admin
user_test_tester = testing .admin
user_test2_tester2 = testing2 .admin
user_test_tester3 = testing3
# To allow accounts/users with underscores you can base64 encode them.
# Here is the account "under_score" and username "a_b" (note the lack
# of padding equal signs):
user64_dW5kZXJfc2NvcmU_YV9i = testing4
```

See the proxy-server.conf-sample for more information.

Account/User List

All accounts/users are listed in the filter section. The format is:

```
user_<account>_<user> = <key> [group] [group] [...] [storage_url]
```

If you want to be able to include underscores in the `<account>` or `<user>` portions, you can base64 encode them (with *no* equal signs) in a line like this:

```
user64_<account_b64>_<user_b64> = <key> [group] [...] [storage_url]
```

There are three special groups:

- `.reseller_admin` can do anything to any account for this auth
- `.reseller_reader` can GET/HEAD anything in any account for this auth
- `.admin` can do anything within the account

If none of these groups are specified, the user can only access containers that have been explicitly allowed for them by a `.admin` or `.reseller_admin`.

The trailing optional `storage_url` allows you to specify an alternate URL to hand back to the user upon authentication. If not specified, this defaults to:

```
$HOST/v1/<reseller_prefix>_<account>
```

Where $HOST will do its best to resolve to what the requester would need to use to reach this host, <reseller_prefix> is from this section, and <account> is from the user_<account>_<user> name. Note that $HOST cannot possibly handle when you have a load balancer in front of it that does https while TempAuth itself runs with http; in such a case, youll have to specify the storage_url_scheme configuration value as an override.

Multiple Reseller Prefix Items

The reseller prefix specifies which parts of the account namespace this middleware is responsible for managing authentication and authorization. By default, the prefix is AUTH so accounts and tokens are prefixed by AUTH_. When a requests token and/or path start with AUTH_, this middleware knows it is responsible.

We allow the reseller prefix to be a list. In tempauth, the first item in the list is used as the prefix for tokens and user groups. The other prefixes provide alternate accounts that users can access. For example if the reseller prefix list is AUTH, OTHER, a user with admin access to AUTH_account also has admin access to OTHER_account.

Required Group

The group .admin is normally needed to access an account (ACLs provide an additional way to access an account). You can specify the require_group parameter. This means that you also need the named group to access an account. If you have several reseller prefix items, prefix the require_group parameter with the appropriate prefix.

X-Service-Token

If an X-Service-Token is presented in the request headers, the groups derived from the token are appended to the roles derived from X-Auth-Token. If X-Auth-Token is missing or invalid, X-Service-Token is not processed.

The X-Service-Token is useful when combined with multiple reseller prefix items. In the following configuration, accounts prefixed SERVICE_ are only accessible if X-Auth-Token is from the end-user and X-Service-Token is from the glance user:

```
[filter:tempauth]
use = egg:swift#tempauth
reseller_prefix = AUTH, SERVICE
SERVICE_require_group = .service
user_admin_admin = admin .admin .reseller_admin
user_joeacct_joe = joepw .admin
user_maryacct_mary = marypw .admin
user_glance_glance = glancepw .service
```

The name .service is an example. Unlike .admin, .reseller_admin, .reseller_reader it is not a reserved name.

Please note that ACLs can be set on service accounts and are matched against the identity validated by `X-Auth-Token`. As such ACLs can grant access to a service accounts container without needing to provide a service token, just like any other cross-reseller request using ACLs.

Account ACLs

If a swift_owner issues a POST or PUT to the account with the `X-Account-Access-Control` header set in the request, then this may allow certain types of access for additional users.

- Read-Only: Users with read-only access can list containers in the account, list objects in any container, retrieve objects, and view unprivileged account/container/object metadata.
- Read-Write: Users with read-write access can (in addition to the read-only privileges) create objects, overwrite existing objects, create new containers, and set unprivileged container/object metadata.
- Admin: Users with admin access are swift_owners and can perform any action, including viewing/setting privileged metadata (e.g. changing account ACLs).

To generate headers for setting an account ACL:

```
from swift.common.middleware.acl import format_acl
acl_data = { 'admin': ['alice'], 'read-write': ['bob', 'carol'] }
header_value = format_acl(version=2, acl_dict=acl_data)
```

To generate a curl command line from the above:

```
token=...
storage_url=...
python -c '
  from swift.common.middleware.acl import format_acl
  acl_data = { 'admin': ['alice'], 'read-write': ['bob', 'carol'] }
  headers = {'X-Account-Access-Control':
             format_acl(version=2, acl_dict=acl_data)}
  header_str = ' '.join(["-H '%s: %s'" % (k, v)
                         for k, v in headers.items()])
  print('curl -D- -X POST -H "x-auth-token: $token" %s '
        '$storage_url' % header_str)
'
```

class swift.common.middleware.tempauth.**TempAuth**(*app*, *conf*)

Bases: `object`

Parameters

- **app** The next WSGI app in the pipeline
- **conf** The dict of configuration values from the Paste config file

account_acls(*req*)

Return a dict of ACL data from the account server via get_account_info.

Auth systems may define their own format, serialization, structure, and capabilities implemented in the ACL headers and persisted in the sysmeta data. However, auth systems are strongly encouraged to be interoperable with Tempauth.

Account ACLs are set and retrieved via the header X-Account-Access-Control

For header format and syntax, see:

- *swift.common.middleware.acl.parse_acl()*
- *swift.common.middleware.acl.format_acl()*

authorize(*req*)

Returns None if the request is authorized to continue or a standard WSGI response callable if not.

denied_response(*req*)

Returns a standard WSGI response callable with the status of 403 or 401 depending on whether the REMOTE_USER is set or not.

extract_acl_and_report_errors(*req*)

Return a user-readable string indicating the errors in the input ACL, or None if there are no errors.

get_groups(*env*, *token*)

Get groups for the given token.

Parameters

- **env** The current WSGI environment dictionary.
- **token** Token to validate and return a group string for.

Returns None if the token is invalid or a string containing a comma separated list of groups the authenticated user is a member of. The first group in the list is also considered a unique identifier for that user.

handle(*env*, *start_response*)

WSGI entry point for auth requests (ones that match the self.auth_prefix). Wraps env in swob.Request object and passes it down.

Parameters

- **env** WSGI environment dictionary
- **start_response** WSGI callable

handle_get_token(*req*)

Handles the various *request for token and service end point(s)* calls. There are various formats to support the various auth servers in the past. Examples:

```
GET <auth-prefix>/v1/<act>/auth
    X-Auth-User: <act>:<usr>  or  X-Storage-User: <usr>
    X-Auth-Key: <key>         or  X-Storage-Pass: <key>
GET <auth-prefix>/auth
    X-Auth-User: <act>:<usr>  or  X-Storage-User: <act>:<usr>
    X-Auth-Key: <key>         or  X-Storage-Pass: <key>
GET <auth-prefix>/v1.0
    X-Auth-User: <act>:<usr>  or  X-Storage-User: <act>:<usr>
    X-Auth-Key: <key>         or  X-Storage-Pass: <key>
```

On successful authentication, the response will have X-Auth-Token and X-Storage-Token set to the token to use with Swift and X-Storage-URL set to the URL to the default Swift cluster to use.

Parameters **req** The swob.Request to process.

Returns swob.Response, 2xx on success with data set as explained above.

handle_request(*req*)

Entry point for auth requests (ones that match the self.auth_prefix). Should return a WSGI-style callable (such as swob.Response).

Parameters **req** swob.Request object

swift.common.middleware.tempauth.**filter_factory**(*global_conf*, ***local_conf*)

Returns a WSGI filter app for use with paste.deploy.

9.8.32 TempURL

TempURL Middleware

Allows the creation of URLs to provide temporary access to objects.

For example, a website may wish to provide a link to download a large object in Swift, but the Swift account has no public access. The website can generate a URL that will provide GET access for a limited time to the resource. When the web browser user clicks on the link, the browser will download the object directly from Swift, obviating the need for the website to act as a proxy for the request.

If the user were to share the link with all his friends, or accidentally post it on a forum, etc. the direct access would be limited to the expiration time set when the website created the link.

Beyond that, the middleware provides the ability to create URLs, which contain signatures which are valid for all objects which share a common prefix. These prefix-based URLs are useful for sharing a set of objects.

Restrictions can also be placed on the ip that the resource is allowed to be accessed from. This can be useful for locking down where the urls can be used from.

Client Usage

To create temporary URLs, first an `X-Account-Meta-Temp-URL-Key` header must be set on the Swift account. Then, an HMAC (RFC 2104) signature is generated using the HTTP method to allow (`GET`, `PUT`, `DELETE`, etc.), the Unix timestamp until which the access should be allowed, the full path to the object, and the key set on the account.

The digest algorithm to be used may be configured by the operator. By default, HMAC-SHA256 and HMAC-SHA512 are supported. Check the `tempurl.allowed_digests` entry in the clusters capabilities response to see which algorithms are supported by your deployment; see *Discoverability* for more information. On older clusters, the `tempurl` key may be present while the `allowed_digests` subkey is not; in this case, only HMAC-SHA1 is supported.

For example, here is code generating the signature for a `GET` for 60 seconds on `/v1/AUTH_account/container/object`:

```
import hmac
from hashlib import sha256
from time import time
method = 'GET'
expires = int(time() + 60)
path = '/v1/AUTH_account/container/object'
key = 'mykey'
hmac_body = '%s\n%s\n%s' % (method, expires, path)
sig = hmac.new(key, hmac_body, sha256).hexdigest()
```

Be certain to use the full path, from the `/v1/` onward.

Lets say `sig` ends up equaling `732fcac368abb10c78a4cbe95c3fab7f311584532bf779abd5074e13cbe8b88b` and `expires` ends up `1512508563`. Then, for example, the website could provide a link to:

```
https://swift-cluster.example.com/v1/AUTH_account/container/object?
temp_url_sig=732fcac368abb10c78a4cbe95c3fab7f311584532bf779abd5074e13cbe8b88b&
temp_url_expires=1512508563
```

For longer hashes, a hex encoding becomes unwieldy. Base64 encoding is also supported, and indicated by prefixing the signature with "`<digest name>:`". This is *required* for HMAC-SHA512 signatures. For example, comparable code for generating a HMAC-SHA512 signature would be:

```
import base64
import hmac
from hashlib import sha512
from time import time
method = 'GET'
expires = int(time() + 60)
path = '/v1/AUTH_account/container/object'
key = 'mykey'
hmac_body = '%s\n%s\n%s' % (method, expires, path)
sig = 'sha512:' + base64.urlsafe_b64encode(hmac.new(
    key, hmac_body, sha512).digest())
```

Supposing that `sig` ends up equaling `sha512:ZrSijn0GyDhsv1ltIj9hWUTrbAeE45NcKXyBaz7aPbSMvROQ4jtYH4nRAmm5ErY2X11Yc1Yhy2OMCyN3yueeXg==` and `expires` ends up `1516741234`, then the website could provide a link to:

```
https://swift-cluster.example.com/v1/AUTH_account/container/object?
temp_url_sig=sha512:ZrSijn0GyDhsv1ltIj9hWUTrbAeE45NcKXyBaz7aPbSMvRO
Q4jtYH4nRAmm5ErY2X11Yc1Yhy2OMCyN3yueeXg==&
temp_url_expires=1516741234
```

You may also use ISO 8601 UTC timestamps with the format "`%Y-%m-%dT%H:%M:%SZ`" instead of UNIX timestamps in the URL (but NOT in the code above for generating the signature!). So, the above HMAC-SHA246 URL could also be formulated as:

```
https://swift-cluster.example.com/v1/AUTH_account/container/object?
temp_url_sig=732fcac368abb10c78a4cbe95c3fab7f311584532bf779abd5074e13cbe8b88b&
temp_url_expires=2017-12-05T21:16:03Z
```

If a prefix-based signature with the prefix `pre` is desired, set path to:

```
path = 'prefix:/v1/AUTH_account/container/pre'
```

The generated signature would be valid for all objects starting with `pre`. The middleware detects a prefix-based temporary URL by a query parameter called `temp_url_prefix`. So, if `sig` and `expires` would end up like above, following URL would be valid:

```
https://swift-cluster.example.com/v1/AUTH_account/container/pre/object?
temp_url_sig=732fcac368abb10c78a4cbe95c3fab7f311584532bf779abd5074e13cbe8b88b&
temp_url_expires=1512508563&
temp_url_prefix=pre
```

Another valid URL:

```
https://swift-cluster.example.com/v1/AUTH_account/container/pre/
subfolder/another_object?
temp_url_sig=732fcac368abb10c78a4cbe95c3fab7f311584532bf779abd5074e13cbe8b88b&
temp_url_expires=1512508563&
temp_url_prefix=pre
```

If you wish to lock down the ip ranges from where the resource can be accessed to the ip `1.2.3.4`:

```
import hmac
from hashlib import sha256
from time import time
method = 'GET'
expires = int(time() + 60)
path = '/v1/AUTH_account/container/object'
ip_range = '1.2.3.4'
key = b'mykey'
hmac_body = 'ip=%s\n%s\n%s\n%s' % (ip_range, method, expires, path)
sig = hmac.new(key, hmac_body.encode('ascii'), sha256).hexdigest()
```

The generated signature would only be valid from the ip `1.2.3.4`. The middleware detects an ip-based temporary URL by a query parameter called `temp_url_ip_range`. So, if `sig` and `expires` would end up like above, following URL would be valid:

```
https://swift-cluster.example.com/v1/AUTH_account/container/object?
temp_url_sig=3f48476acaf5ec272acd8e99f7b5bad96c52ddba53ed27c60613711774a06f0c&
temp_url_expires=1648082711&
temp_url_ip_range=1.2.3.4
```

Similarly to lock down the ip to a range of `1.2.3.X` so starting from the ip `1.2.3.0` to `1.2.3.255`:

```
import hmac
from hashlib import sha256
from time import time
method = 'GET'
expires = int(time() + 60)
path = '/v1/AUTH_account/container/object'
ip_range = '1.2.3.0/24'
```

(continues on next page)

(continued from previous page)

```
key = b'mykey'
hmac_body = 'ip=%s\n%s\n%s\n%s' % (ip_range, method, expires, path)
sig = hmac.new(key, hmac_body.encode('ascii'), sha256).hexdigest()
```

Then the following url would be valid:

```
https://swift-cluster.example.com/v1/AUTH_account/container/object?
temp_url_sig=6ff81256b8a3ba11d239da51a703b9c06a56ffddeb8caab74ca83af8f73c9c83&
temp_url_expires=1648082711&
temp_url_ip_range=1.2.3.0/24
```

Any alteration of the resource path or query arguments of a temporary URL would result in `401 Unauthorized`. Similarly, a `PUT` where `GET` was the allowed method would be rejected with `401 Unauthorized`. However, `HEAD` is allowed if `GET`, `PUT`, or `POST` is allowed.

Using this in combination with browser form post translation middleware could also allow direct-from-browser uploads to specific locations in Swift.

TempURL supports both account and container level keys. Each allows up to two keys to be set, allowing key rotation without invalidating all existing temporary URLs. Account keys are specified by `X-Account-Meta-Temp-URL-Key` and `X-Account-Meta-Temp-URL-Key-2`, while container keys are specified by `X-Container-Meta-Temp-URL-Key` and `X-Container-Meta-Temp-URL-Key-2`. Signatures are checked against account and container keys, if present.

With `GET` TempURLs, a `Content-Disposition` header will be set on the response so that browsers will interpret this as a file attachment to be saved. The filename chosen is based on the object name, but you can override this with a filename query parameter. Modifying the above example:

```
https://swift-cluster.example.com/v1/AUTH_account/container/object?
temp_url_sig=732fcac368abb10c78a4cbe95c3fab7f311584532bf779abd5074e13cbe8b88b&
temp_url_expires=1512508563&filename=My+Test+File.pdf
```

If you do not want the object to be downloaded, you can cause `Content-Disposition: inline` to be set on the response by adding the `inline` parameter to the query string, like so:

```
https://swift-cluster.example.com/v1/AUTH_account/container/object?
temp_url_sig=732fcac368abb10c78a4cbe95c3fab7f311584532bf779abd5074e13cbe8b88b&
temp_url_expires=1512508563&inline
```

In some cases, the client might not able to present the content of the object, but you still want the content able to save to local with the specific filename. So you can cause `Content-Disposition: inline; filename=...` to be set on the response by adding the `inline&filename=...` parameter to the query string, like so:

```
https://swift-cluster.example.com/v1/AUTH_account/container/object?
temp_url_sig=732fcac368abb10c78a4cbe95c3fab7f311584532bf779abd5074e13cbe8b88b&
temp_url_expires=1512508563&inline&filename=My+Test+File.pdf
```

Cluster Configuration

This middleware understands the following configuration settings:

incoming_remove_headers A whitespace-delimited list of the headers to remove from incoming requests. Names may optionally end with * to indicate a prefix match. `incoming_allow_headers` is a list of exceptions to these removals. Default: `x-timestamp`

incoming_allow_headers A whitespace-delimited list of the headers allowed as exceptions to `incoming_remove_headers`. Names may optionally end with * to indicate a prefix match.

Default: None

outgoing_remove_headers A whitespace-delimited list of the headers to remove from outgoing responses. Names may optionally end with * to indicate a prefix match. `outgoing_allow_headers` is a list of exceptions to these removals.

Default: `x-object-meta-*`

outgoing_allow_headers A whitespace-delimited list of the headers allowed as exceptions to `outgoing_remove_headers`. Names may optionally end with * to indicate a prefix match.

Default: `x-object-meta-public-*`

methods A whitespace delimited list of request methods that are allowed to be used with a temporary URL.

Default: `GET HEAD PUT POST DELETE`

allowed_digests A whitespace delimited list of digest algorithms that are allowed to be used when calculating the signature for a temporary URL.

Default: `sha256 sha512`

`swift.common.middleware.tempurl.`**`DEFAULT_INCOMING_ALLOW_HEADERS = ''`**

Default headers as exceptions to DEFAULT_INCOMING_REMOVE_HEADERS. Simply a whitespace delimited list of header names and names can optionally end with * to indicate a prefix match.

`swift.common.middleware.tempurl.`**`DEFAULT_INCOMING_REMOVE_HEADERS = 'x-timestamp'`**

Default headers to remove from incoming requests. Simply a whitespace delimited list of header names and names can optionally end with * to indicate a prefix match. DEFAULT_INCOMING_ALLOW_HEADERS is a list of exceptions to these removals.

`swift.common.middleware.tempurl.`**`DEFAULT_OUTGOING_ALLOW_HEADERS = 'x-object-meta-public-*'`**

Default headers as exceptions to DEFAULT_OUTGOING_REMOVE_HEADERS. Simply a whitespace delimited list of header names and names can optionally end with * to indicate a prefix match.

`swift.common.middleware.tempurl.`**`DEFAULT_OUTGOING_REMOVE_HEADERS = 'x-object-meta-*'`**

Default headers to remove from outgoing responses. Simply a whitespace delimited list of header names and names can optionally end with * to indicate a prefix match. DEFAULT_OUTGOING_ALLOW_HEADERS is a list of exceptions to these removals.

class swift.common.middleware.tempurl.**TempURL**(*app*, *conf*, *logger=None*)

Bases: object

WSGI Middleware to grant temporary URLs specific access to Swift resources. See the overview for more information.

The proxy logs created for any subrequests made will have swift.source set to TU.

Parameters

- **app** The next WSGI filter or app in the paste.deploy chain.
- **conf** The configuration dict for the middleware.

agent

HTTP user agent to use for subrequests.

app

The next WSGI application/filter in the paste.deploy pipeline.

conf

The filter configuration dict.

incoming_allow_headers

Headers to allow in incoming requests. Uppercase WSGI env style, like *HTTP_X_MATCHES_REMOVE_PREFIX_BUT_OKAY*.

incoming_allow_headers_startswith

Header with match prefixes to allow in incoming requests. Uppercase WSGI env style, like *HTTP_X_MATCHES_REMOVE_PREFIX_BUT_OKAY_**.

incoming_remove_headers

Headers to remove from incoming requests. Uppercase WSGI env style, like *HTTP_X_PRIVATE*.

incoming_remove_headers_startswith

Header with match prefixes to remove from incoming requests. Uppercase WSGI env style, like *HTTP_X_SENSITIVE_**.

outgoing_allow_headers

Headers to allow in outgoing responses. Lowercase, like *x-matches-remove-prefix-but-okay*.

outgoing_allow_headers_startswith

Header with match prefixes to allow in outgoing responses. Lowercase, like *x-matches-remove-prefix-but-okay-**.

outgoing_remove_headers

Headers to remove from outgoing responses. Lowercase, like *x-account-meta-temp-url-key*.

outgoing_remove_headers_startswith

Header with match prefixes to remove from outgoing responses. Lowercase, like *x-account-meta-private-**.

swift.common.middleware.tempurl.**filter_factory**(*global_conf*, ***local_conf*)

Returns the WSGI filter for use with paste.deploy.

9.8.33 Versioned Writes

Note: This middleware supports two legacy modes of object versioning that is now replaced by a new mode. It is recommended to use the new *Object Versioning* mode for new containers.

Object versioning in swift is implemented by setting a flag on the container to tell swift to version all objects in the container. The value of the flag is the URL-encoded container name where the versions are stored (commonly referred to as the archive container). The flag itself is one of two headers, which determines how object `DELETE` requests are handled:

- `X-History-Location`

  On `DELETE`, copy the current version of the object to the archive container, write a zero-byte delete marker object that notes when the delete took place, and delete the object from the versioned container. The object will no longer appear in container listings for the versioned container and future requests there will return `404 Not Found`. However, the content will still be recoverable from the archive container.

- `X-Versions-Location`

  On `DELETE`, only remove the current version of the object. If any previous versions exist in the archive container, the most recent one is copied over the current version, and the copy in the archive container is deleted. As a result, if you have 5 total versions of the object, you must delete the object 5 times for that object name to start responding with `404 Not Found`.

Either header may be used for the various containers within an account, but only one may be set for any given container. Attempting to set both simulataneously will result in a `400 Bad Request` response.

Note: It is recommended to use a different archive container for each container that is being versioned.

Note: Enabling versioning on an archive container is not recommended.

When data is `PUT` into a versioned container (a container with the versioning flag turned on), the existing data in the file is redirected to a new object in the archive container and the data in the `PUT` request is saved as the data for the versioned object. The new object name (for the previous version) is `<archive_container>/<length><object_name>/<timestamp>`, where `length` is the 3-character zero-padded hexadecimal length of the `<object_name>` and `<timestamp>` is the timestamp of when the previous version was created.

A `GET` to a versioned object will return the current version of the object without having to do any request redirects or metadata lookups.

A `POST` to a versioned object will update the object metadata as normal, but will not create a new version of the object. In other words, new versions are only created when the content of the object changes.

A `DELETE` to a versioned object will be handled in one of two ways, as described above.

To restore a previous version of an object, find the desired version in the archive container then issue a `COPY` with a `Destination` header indicating the original location. This will archive the current version similar to a `PUT` over the versioned object. If the client additionally wishes to permanently delete what was the current version, it must find the newly-created archive in the archive container and issue a separate `DELETE` to it.

How to Enable Object Versioning in a Swift Cluster

This middleware was written as an effort to refactor parts of the proxy server, so this functionality was already available in previous releases and every attempt was made to maintain backwards compatibility. To allow operators to perform a seamless upgrade, it is not required to add the middleware to the proxy pipeline and the flag `allow_versions` in the container server configuration files are still valid, but only when using `X-Versions-Location`. In future releases, `allow_versions` will be deprecated in favor of adding this middleware to the pipeline to enable or disable the feature.

In case the middleware is added to the proxy pipeline, you must also set `allow_versioned_writes` to `True` in the middleware options to enable the information about this middleware to be returned in a /info request.

Note: You need to add the middleware to the proxy pipeline and set `allow_versioned_writes = True` to use `X-History-Location`. Setting `allow_versions = True` in the container server is not sufficient to enable the use of `X-History-Location`.

Upgrade considerations

If `allow_versioned_writes` is set in the filter configuration, you can leave the `allow_versions` flag in the container server configuration files untouched. If you decide to disable or remove the `allow_versions` flag, you must re-set any existing containers that had the `X-Versions-Location` flag configured so that it can now be tracked by the versioned_writes middleware.

Clients should not use the `X-History-Location` header until all proxies in the cluster have been upgraded to a version of Swift that supports it. Attempting to use `X-History-Location` during a rolling upgrade may result in some requests being served by proxies running old code, leading to data loss.

Examples Using `curl` with `X-Versions-Location`

First, create a container with the `X-Versions-Location` header or add the header to an existing container. Also make sure the container referenced by the `X-Versions-Location` exists. In this example, the name of that container is versions:

```
curl -i -XPUT -H "X-Auth-Token: <token>" -H "X-Versions-Location: versions"␣
↪http://<storage_url>/container
curl -i -XPUT -H "X-Auth-Token: <token>" http://<storage_url>/versions
```

Create an object (the first version):

```
curl -i -XPUT --data-binary 1 -H "X-Auth-Token: <token>" http://<storage_url>/
↪container/myobject
```

Now create a new version of that object:

```
curl -i -XPUT --data-binary 2 -H "X-Auth-Token: <token>" http://<storage_url>/
↪container/myobject
```

See a listing of the older versions of the object:

```
curl -i -H "X-Auth-Token: <token>" http://<storage_url>/versions?
↪prefix=008myobject/
```

Now delete the current version of the object and see that the older version is gone from versions container and back in container container:

```
curl -i -XDELETE -H "X-Auth-Token: <token>" http://<storage_url>/container/
↪myobject
curl -i -H "X-Auth-Token: <token>" http://<storage_url>/versions?
↪prefix=008myobject/
curl -i -XGET -H "X-Auth-Token: <token>" http://<storage_url>/container/
↪myobject
```

Examples Using `curl` with `X-History-Location`

As above, create a container with the `X-History-Location` header and ensure that the container referenced by the `X-History-Location` exists. In this example, the name of that container is versions:

```
curl -i -XPUT -H "X-Auth-Token: <token>" -H "X-History-Location: versions"␣
↪http://<storage_url>/container
curl -i -XPUT -H "X-Auth-Token: <token>" http://<storage_url>/versions
```

Create an object (the first version):

```
curl -i -XPUT --data-binary 1 -H "X-Auth-Token: <token>" http://<storage_url>/
↪container/myobject
```

Now create a new version of that object:

```
curl -i -XPUT --data-binary 2 -H "X-Auth-Token: <token>" http://<storage_url>/
↪container/myobject
```

Now delete the current version of the object. Subsequent requests will 404:

```
curl -i -XDELETE -H "X-Auth-Token: <token>" http://<storage_url>/container/
↪myobject
curl -i -H "X-Auth-Token: <token>" http://<storage_url>/container/myobject
```

A listing of the older versions of the object will include both the first and second versions of the object, as well as a delete marker object:

```
curl -i -H "X-Auth-Token: <token>" http://<storage_url>/versions?
↪prefix=008myobject/
```

To restore a previous version, simply `COPY` it from the archive container:

```
curl -i -XCOPY -H "X-Auth-Token: <token>" http://<storage_url>/versions/
↪008myobject/<timestamp> -H "Destination: container/myobject"
```

Note that the archive container still has all previous versions of the object, including the source for the restore:

```
curl -i -H "X-Auth-Token: <token>" http://<storage_url>/versions?
↪prefix=008myobject/
```

To permanently delete a previous version, DELETE it from the archive container:

```
curl -i -XDELETE -H "X-Auth-Token: <token>" http://<storage_url>/versions/
↪008myobject/<timestamp>
```

How to Disable Object Versioning in a Swift Cluster

If you want to disable all functionality, set `allow_versioned_writes` to `False` in the middleware options.

Disable versioning from a container (x is any value except empty):

```
curl -i -XPOST -H "X-Auth-Token: <token>" -H "X-Remove-Versions-Location: x"␣
↪http://<storage_url>/container
```

class swift.common.middleware.versioned_writes.legacy.**VersionedWritesContext**(*wsgi_app*, *logger*)

Bases: *swift.common.wsgi.WSGIContext*

handle_obj_versions_delete_pop(*req*, *versions_cont*, *api_version*, *account_name*, *container_name*, *object_name*)

Handle DELETE requests when in stack mode.

Delete current version of object and pop previous version in its place.

Parameters

- **req** original request.
- **versions_cont** container where previous versions of the object are stored.
- **api_version** api version.
- **account_name** account name.
- **container_name** container name.
- **object_name** object name.

handle_obj_versions_delete_push(*req*, *versions_cont*, *api_version*, *account_name*, *container_name*, *object_name*)

Handle DELETE requests when in history mode.

Copy current version of object to versions_container and write a delete marker before proceeding with original request.

Parameters

- **req** original request.
- **versions_cont** container where previous versions of the object are stored.
- **api_version** api version.

- **account_name** account name.
- **object_name** name of object of original request

handle_obj_versions_put(*req*, *versions_cont*, *api_version*, *account_name*, *object_name*)

Copy current version of object to versions_container before proceeding with original request.

Parameters

- **req** original request.
- **versions_cont** container where previous versions of the object are stored.
- **api_version** api version.
- **account_name** account name.
- **object_name** name of object of original request

9.8.34 XProfile

Profiling middleware for Swift Servers.

The current implementation is based on eventlet aware profiler.(For the future, more profilers could be added in to collect more data for analysis.) Profiling all incoming requests and accumulating cpu timing statistics information for performance tuning and optimization. An mini web UI is also provided for profiling data analysis. It can be accessed from the URL as below.

Index page for browse profile data:

```
http://SERVER_IP:PORT/__profile__
```

List all profiles to return profile ids in json format:

```
http://SERVER_IP:PORT/__profile__/
http://SERVER_IP:PORT/__profile__/all
```

Retrieve specific profile data in different formats:

```
http://SERVER_IP:PORT/__profile__/PROFILE_ID?format=[default|json|csv|ods]
http://SERVER_IP:PORT/__profile__/current?format=[default|json|csv|ods]
http://SERVER_IP:PORT/__profile__/all?format=[default|json|csv|ods]
```

Retrieve metrics from specific function in json format:

```
http://SERVER_IP:PORT/__profile__/PROFILE_ID/NFL?format=json
http://SERVER_IP:PORT/__profile__/current/NFL?format=json
http://SERVER_IP:PORT/__profile__/all/NFL?format=json

NFL is defined by concatenation of file name, function name and the first
line number.
e.g.::
    account.py:50(GETorHEAD)
or with full path:
    opt/stack/swift/swift/proxy/controllers/account.py:50(GETorHEAD)
```

(continues on next page)

(continued from previous page)

```
A list of URL examples:

http://localhost:8080/__profile__     (proxy server)
http://localhost:6200/__profile__/all     (object server)
http://localhost:6201/__profile__/current     (container server)
http://localhost:6202/__profile__/12345?format=json     (account server)
```

The profiling middleware can be configured in paste file for WSGI servers such as proxy, account, container and object servers. Please refer to the sample configuration files in etc directory.

The profiling data is provided with four formats such as binary(by default), json, csv and odf spreadsheet which requires installing odfpy library:

```
sudo pip install odfpy
```

Theres also a simple visualization capability which is enabled by using matplotlib toolkit. it is also required to be installed if you want to use it to visualize statistic data:

```
sudo apt-get install python-matplotlib
```

9.9 Object Audit Watchers

9.9.1 Dark Data

The name of Dark Data refers to the scientific hypothesis of Dark Matter, which supposes that the universe contains a lot of matter than we cannot observe. The Dark Data in Swift is the name of objects that are not accounted in the containers.

The experience of running large scale clusters suggests that Swift does not have any particular bugs that trigger creation of dark data. So, this is an excercise in writing watchers, with a plausible function.

When enabled, Dark Data watcher definitely drags down the clusters overall performance. Of course, the load increase can be mitigated as usual, but at the expense of the total time taken by the pass of auditor.

Because the watcher only deems an object dark when all container servers agree, it will silently fail to detect anything if even one of container servers in the ring is down or unreacheable. This is done in the interest of operators who run with action=delete.

Finally, keep in mind that Dark Data watcher needs the container ring to operate, but runs on an object node. This can come up if cluster has nodes separated by function.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

c

o

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z