# php-opencloud Documentation

***Release 1.12.1***

**Jamie Hannaford, Shaunak Kashyap**

**Jul 18, 2017**

# Contents

CHAPTER 1

# Installation

You must install this library through Composer:

```
composer require rackspace/php-opencloud
```

If you do not have Composer installed, please consult the official docs.

Once you have installed the library, you will need to load Composer's autoloader (which registers all the required namespaces). To do this, place the following line of PHP code at the top of your application's PHP files:

```
require 'vendor/autoload.php';
```

This assumes your application's PHP files are located in the same folder as `vendor/`. If your files are located elsewhere, please supply the path to `vendor/autoload.php` in the require statement above.

Read the getting-started-with-openstack or getting-started-with-rackspace to help you get started with basic Compute operations.

**Note:** If you are running PHP 5.3, please see our using-php-5.3 guide.

CHAPTER 2

# Services

## Auto Scale v2

**Note:** This service is only available for Rackspace users.

### Setup

The first step is to pass in your credentials and set up a client. For Rackspace users, you will need your username and API key:

```
use OpenCloud\Rackspace;

$client = new Rackspace(Rackspace::US_IDENTITY_ENDPOINT, array(
  'username' => '{username}',
  'apiKey'   => '{apiKey}',
));
```

#### Auto Scale service

Now to instantiate the Auto Scale service:

```
$service = $client->autoscaleService();
```

- `{catalogName}` is the name of the service as it appears in the service catalog. OpenStack users *must* set this value. For Rackspace users, a default will be provided if you pass in `null`.
- `{region}` is the region the service will operate in. For Rackspace users, you can select one of the following from the *supported regions page*.

- `{urlType}` is the *type of URL* to use, depending on which endpoints your catalog provides. If omitted, it will default to the public network.

## Operations

### Groups

### List all groups

```
$groups = $service->groupList();
foreach ($group as $group) {
  /** @var $group OpenCloud\Autoscale\Resources\Group */
}
```

Please consult the iterator guide for more information about iterators.

### Retrieve group by ID

```
$group = $service->group('{groupId}');
```

### Create a new group

```
// Set the config object for this autoscale group; contains all of properties
// which determine its behaviour
$groupConfig = array(
  'name'        => 'new_autoscale_group',
  'minEntities' => 5,
  'maxEntities' => 25,
  'cooldown'    => 60,
);

// We need specify what is going to be launched. For now, we'll launch a new server
$launchConfig = array(
  'type' => 'launch_server',
  'args' => array(
    'server' => array(
      'flavorRef' => 3,
      'name'      => 'webhead',
      'imageRef'  => '0d589460-f177-4b0f-81c1-8ab8903ac7d8'
    ),
    'loadBalancers' => array(
      array('loadBalancerId' => 2200, 'port' => 8081),
    )
  )
);

// Do we want particular scaling policies?
$policy = array(
  'name'     => 'scale up by 10',
  'change'   => 10,
  'cooldown' => 5,
  'type'     => 'webhook',
```

```
);

$group->create(array(
  'groupConfiguration'  => $groupConfig,
  'launchConfiguration' => $launchConfig,
  'scalingPolicies'     => array($policy),
));
```

### Delete a group

```
$group->delete();
```

### Get the current state of the scaling group

```
$group->getState();
```

### Group configurations

### Setup

In order to interact with the functionality of a group's configuration, you must first retrieve the details of the group itself. To do this, you must substitute *{groupId}* for your group's ID:

```
$group = $service->group('{groupId}');
```

### Get group configuration

```
/** @var  */
$groupConfig = $group->getGroupConfig();
```

### Edit group configuration

```
$groupConfig->update(array(
    'name' => 'New name!'
));
```

### Get launch configuration

```
/** @var */
$launchConfig = $group->getLaunchConfig();
```

### Edit group/launch configuration

```
$launchConfig = $group->getLaunchConfig();

$server = $launchConfig->args->server;
$server->name = "BRAND NEW SERVER NAME";

$launchConfig->update(array
    'args' => array(
        'server' => $server,
        'loadBalancers' => $launchConfig->args->loadBalancers
    )
));
```

### Scaling Policies

### Setup

In order to interact with the functionality of a group's scaling policies, you must first retrieve the details of the group itself. To do this, you must substitute *{groupId}* for your group's ID:

```
$group = $service->group('{groupId}');
```

### Get all policies

```
$policies = $group->getScalingPolicies();

foreach ($policies as $policy) {
    printf("Name: %s Type: %s\n", $policy->name, $policy->type);
}
```

### Create new scaling policies

Creating policies is achieved through passing an array to the `create` method.

```
$policies = array(
  array(
    'name'     => 'NEW NAME',
    'change'   => 1,
    'cooldown' => 150,
    'type'     => 'webhook',
  )
);

$group->createScalingPolicies($policies);
```

### Get an existing scaling policy

```
$policy = $group->getScalingPolicy('{policyId}');
```

### Update a scaling policy

```
$policy = $group->getScalingPolicy('{policyId}');
$policy->update(array(
    'name' => 'More relevant name'
));
```

### Delete a scaling policy

```
$policy = $group->getScalingPolicy('{policyId}');
$policy->delete();
```

### Execute a scaling policy

```
$policy = $group->getScalingPolicy('{policyId}');
$policy->execute();
```

### Webhooks

### Setup

In order to interact with webhooks, you must first retrieve the details of the group and scaling policy you want to execute:

```
$group = $service->group('{groupId}');
$policy = $group->getScalingPolicy('{policyId}');
```

### Get all webhooks

```
$webhooks = $policy->getWebookList();
```

### Create a new webhook

```
$policy->createWebhooks(array(
    array(
        'name' => 'Alice',
        'metadata' => array(
            'firstKey'  => 'foo',
            'secondKey' => 'bar'
        )
    )
));
```

#### Get webhook

```
$webhook = $policy->getWebhook('{webhookId}');
```

#### Update webhook

```
// Update the metadata
$metadata = $webhook->metadata;
$metadata->thirdKey = 'blah';
$webhook->update(array(
    'metadata' => $metadata
));
```

#### Delete webhook

### Glossary

**group** The scaling group is at the heart of an Auto Scale deployment. The scaling group specifies the basic elements of the Auto Scale configuration. It manages how many servers can participate in the scaling group. It also specifies information related to load balancers if your configuration uses a load balancer.

**group configuration** Outlines the basic elements of the Auto Scale configuration. The group configuration manages how many servers can participate in the scaling group. It sets a minimum and maximum limit for the number of entities that can be used in the scaling process. It also specifies information related to load balancers.

**launch configuration** Creates a blueprint for how new servers will be created. The launch configuration specifies what type of server image will be started on launch, what flavor the new server is, and which load balancer the new server connects to.

**policy** Auto Scale uses policies to define the scaling activity that will take place, as well as when and how that scaling activity will take place. Scaling policies specify how to modify the scaling group and its behavior. You can specify multiple policies to manage a scaling group.

**webhook** A webhook is a reachable endpoint that when visited will execute a scaling policy for a particular scaling group.

### Further Links

- Getting Started Guide for the API
- API Developer Guide
- API release history

## Compute v2

### Setup

### Rackspace setup

The first step is to pass in your credentials and set up a client. For Rackspace users, you will need your username and API key:

```
use OpenCloud\Rackspace;

$client = new Rackspace(Rackspace::US_IDENTITY_ENDPOINT, array(
  'username' => '{username}',
  'apiKey'   => '{apiKey}',
));
```

### OpenStack setup

If you're an OpenStack user, you will also need to prove a few other configuration parameters:

```
$client = new OpenCloud\OpenStack('{keystoneUrl}', array(
  'username' => '{username}',
  'password' => '{apiKey}',
  'tenantId' => '{tenantId}',
));
```

### Compute service

Now to instantiate the Compute service:

```
$service = $client->computeService('{catalogName}', '{region}', '{urlType}');
```

- `{catalogName}` is the name of the service as it appears in the service catalog. OpenStack users *must* set this value. For Rackspace users, a default will be provided if you pass in `null`.
- `{region}` is the region the service will operate in. For Rackspace users, you can select one of the following from the *supported regions page*.
- `{urlType}` is the *type of URL* to use, depending on which endpoints your catalog provides. If omitted, it will default to the public network.

## Operations

### Images

---

**Note:** **Images on Rackspace servers:** with standard servers, the entire disk (OS and data) is captured in the image. With Performance servers, only the s ystem disk is captured in the image. The data disks should be backed up using Cloud Backup or Cloud Block Storage to ensure availability in case you need to rebuild or restore a server.

---

### List images

Below is the simplest usage for retrieving a list of images:

```
$images = $service->imageList();

foreach ($images as $image) {

}
```

Get the executable PHP script for this example

### Detailed results

By default, the only fields returned in a list call are *id* and *name*, but you can enable more detailed information to be result by passing in *true* as the first argument of the call, like so:

```
$images = $service->imageList(true);
```

### Filtering

You can also refine the list of images returned by providing specific filters:

| Array key | Description |
|---|---|
| server | Filters the list of images by server. Specify the server reference by ID or by full URL. |
| name | Filters the list of images by image name. |
| status | Filters the list of images by status. In-flight images have a status of `SAVING` and the conditional progress element contains a value from 0 to 100, which indicates the percentage completion. For a full list, please consult the `OpenCloud\Compute\Constants\ImageState` class. Images with an `ACTIVE` status are available for use. |
| changes-since | Filters the list of images to those that have changed since the changes-since time. See the official docs for more information. |
| marker | The ID of the last item in the previous list. See the official docs for more information. |
| limit | Sets the page size. See the official docs for more information. |
| type | Filters base Rackspace images or any custom server images that you have created. Can either be `BASE` or `SNAPSHOT`. |

These are defined in an array and passed in as the second argument. For example, to filter images for a particular server:

```
$images = $service->imageList(false, array('server' => '{serverId}'));
```

### Retrieve details about an image

```
$image = $service->image('{imageId}');
```

### Delete an image

```
$image->delete();
```

### Flavors

### Get a flavor

```
$flavor = $service->flavor('{flavorId}');
```

### List flavors

```
$flavors = $service->flavorList();

foreach ($flavors as $flavor) {
    /** @param $flavor OpenCloud\Common\Resource\FlavorInterface */
}
```

Get the executable PHP script for this example

### Detailed results

By default, the `flavorList` method returns full details on all flavors. However, because of the overhead involved in retrieving all the details, this function can be slower than might be expected. To disable this feature and keep bandwidth at a minimum, just pass `false` as the first argument:

```
// Name and ID only
$compute->flavorList(false);
```

### Filtering

You can also refine the list of images returned by providing specific filters:

| Array key | Description |
|---|---|
| minDisk | Filters the list of flavors to those with the specified minimum number of gigabytes of disk storage. |
| minRam | Filters the list of flavors to those with the specified minimum amount of RAM in megabytes. |
| marker | The ID of the last item in the previous list. See the official docs for more information. |
| limit | Sets the page size. See the official docs for more information. |

These are defined in an array and passed in as the second argument. For example, to return all flavors over 4GB in RAM:

```
$flavors = $service->flavorList(true, array('minRam' => 4));
```

### Servers

### Get server

The easiest way to retrieve a specific server is by its unique ID:

```
$server = $service->server('{serverId}');
```

### List servers

You can list servers in two different ways:

- return an *overview* of each server (ID, name and links)
- return *detailed information* for each server

Knowing which option to use might help save unnecessary bandwidth and reduce latency.

```
// overview
$servers = $service->serverList();

// detailed
$servers = $service->serverList(true);
```

### URL parameters for filtering servers

| Name | Description | Type |
|---|---|---|
| image | The image ID | string |
| flavor | The flavor ID | string |
| name | The server name | string |
| status | The server status. Servers contain a status attribute that indicates the current server state. You can filter on the server status when you complete a list servers request, and the server status is returned in the response body. For a full list, please consult `OpenCloud\Compute\Constants\ServerState` | string |
| changes-since | Value for checking for changes since a previous request | A valid ISO 8601 dateTime (2011-01-24T17:08Z) |
| RAX-SI:image_schedule | If scheduled images enabled or not. If the value is TRUE, the list contains all servers that have an image schedule resource set on them. If the value is set to FALSE, the list contains all servers that do not have an image schedule. | bool |

Get the executable PHP script for this example

### Create server

### Using an image

Now we're ready to create our instance:

```
$server = $compute->server();

$server->create(array(
    'name'     => 'My lovely server',
    'imageId'  => '{imageId}',
    'flavorId' => '{flavorId}',
));
```

It's always best to be defensive when executing functionality over HTTP; you can achieve this best by wrapping calls in a try/catch block. It allows you to debug your failed operations in a graceful and efficient manner.

Get the executable PHP script for this example

### Using a bootable volume

Firstly we need to find our volume using their IDs.

```
$bootableVolume = $client->volumeService()->volume('{volumeId}');
```

Now we're ready to create our instance:

```
$server = $compute->server();

$response = $server->create(array(
    'name'      => 'My lovely server',
    'volume'    => $bootableVolume,
    'flavorId' => '{flavorId}'
));
```

It's always best to be defensive when executing functionality over HTTP; you can achieve this best by wrapping calls in a try/catch block. It allows you to debug your failed operations in a graceful and efficient manner.

Get the executable PHP script for this example

### Create parameters

| Name | Description | Type | Required |
|---|---|---|---|
| name | The server name. The name that you specify in a create request becomes the initial host name of the server. After the server is built, if you change the server name in the API or change the host name directly, the names are not kept in sync. | string | Yes |
| flavor | A populated `OpenCloud\Compute\Resource\Flavor` object representing your chosen flavor | object | Yes |
| image | A populated `OpenCloud\Compute\Resource\Image` object representing your chosen image | object | No, if volume is specified |
| volume | A populated `OpenCloud\Volume\Resource\Volume` object representing your chosen bootable volume | object | No, if image is specified |
| volumeDeleteOnTermination | `true` if the bootable volume should be deleted when the server is terminated; `false`, otherwise | boolean | No; default = `false` |
| OS-DCF:diskConfig | The disk configuration value. You can use two options: `AUTO` or `MANUAL`. `AUTO` means the server is built with a single partition the size of the target flavor disk. The file system is automatically adjusted to fit the entire partition. This keeps things simple and automated. `AUTO` is valid only for images and servers with a single partition that use the EXT3 file system. This is the default setting for applicable Rackspace base images.`MANUAL` means the server is built using whatever partition scheme and file system is in the source image. If the target flavor disk is larger, the remaining disk space is left unpartitioned. This enables images to have non-EXT3 file systems, multiple partitions, and so on, and enables you to manage the disk configuration. | string | No |
| networks | An array of populated `OpenCloud\Compute\Resource\Network` objects that indicate which networks your instance resides in. | array | No |
| metadata | An array of arbitrary data (key-value pairs) that adds additional meaning to your server. | array | No |
| keypair | You can install a registered keypair onto your newly created instance, thereby providing scope for keypair-based authentication. | array | No |
| personality | Files that you can upload to your newly created instance's filesystem. | array | No |
| user_data | User script to configure the server at launch time | string | No |

### Creating a server with keypairs

In order to provision an instance with a saved keypair (allowing you to SSH in without passwords), you create your server using the same operation as usual, with one extra parameter:

```
$server = $compute->server();

$server->create(array(
    'name'     => 'New server',
    'imageId'  => '{imageId}',
    'flavorId' => '{flavorId}',
    'keypair'  => 'main_key'
));
```

So, as you can see, you specify the **name** of an existing keypair that you previously created on the API.

Get the executable PHP script for this example

### Creating a server with personality files

Before you execute the create operation, you can add "personality" files to your `OpenCloud\Compute\Resource\Server` object. These files are structured as a flat array.

```
$server->addFile('/var/test_file', 'FILE CONTENT');
```

As you can see, the first parameter represents the filename, and the second is a string representation of its content. When the server is created these files will be created on its local filesystem. For more information about server personality files, please consult the official documentation.

### Update server

You can update certain attributes of an existing server instance. These attributes are detailed in the next section.

```
$server->update(array(
    'name' => 'NEW SERVER NAME'
));
```

Get the executable PHP script for this example

### Updatable attributes

| name | description |
|---|---|
| name | The name of the server. If you edit the server name, the server host name does not change. Also, server names are not guaranteed to be unique. |
| accessIPv4 | The IP version 4 address. |
| accessIPv6 | The IP version 6 address. |

### Updating the access IP address(es)

For example, you may have a private cloud with internal addresses in the 10.1.x range. However, you can access a server via a firewall device at address 50.57.94.244. In this case, you can change the `accessIPv4` attribute to point to the firewall:

```
$server->update(array('accessIPv4' => '50.57.94.244'));
```

When a client application retrieves the server's information, it will know that it needs to use the `accessIPv4` address to connect to the server, and *not* the IP address assigned to one of the network interfaces.

### Retrieving the server's IP address

The `Server::ip()` method is used to retrieve the server's IP address. It has one optional parameter: the format (either IPv4 or IPv6) of the address to return (by default, it returns the IPv4 address):

```
// IPv4
echo $server->ip();
echo $server->ip(4);

// IPv6
echo $server->ip(6);
```

### Delete server

```
$server->delete();
```

Get the executable PHP script for this example

### Keypairs

### Generate a new keypair

This operation creates a new keypair under a provided name; the public key value is automatically generated for you.

```
// Instantiate empty object
$keypair = $service->keypair();

// Send to API
$keypair->create(array(
    'name' => 'jamie_keypair_1'
));

// Save these!
$pubKey = $keypair->getPublicKey();
$priKey = $keypair->getPrivateKey();
```

Get the executable PHP script for this example

### Upload existing keypair

This operation creates a new keypair according to a provided name and public key value. This is useful when the public key already exists on your local filesystem.

```
$keypair = $service->keypair();

// $key needs to be the string content of the key file, not the filename
$content = file_get_contents('~/.ssh/id_rsa.pub');

$keypair->create(array(
    'name'      => 'main_key',
    'publicKey' => $content
));
```

Get the executable PHP script for this example

#### List keypairs

To list all existing keypairs:

```
$keys = $service->listKeypairs();

foreach ($keys as $key) {

}
```

#### Delete keypairs

To delete a specific keypair:

```
$keypair->delete();
```

### Glossary

**image** An image is a collection of files for a specific operating system that you use to create or rebuild a server. Rackspace provides prebuilt images. You can also create custom images from servers that you have launched.

**flavor** A flavor is a named definition of certain server parameters such as the amount of RAM and disk space available. (There are other parameters set via the flavor, such as the amount of disk space and the number of virtual CPUs, but a discussion of those is too in-depth for a simple Getting Started Guide like this one.)

**server** A server is a virtual machine instance in the Cloud Servers environment.

### Further Links

- Getting Started Guide for the API
- API Developer Guide
- API release history

## Databases v1

### Setup

#### Rackspace setup

The first step is to pass in your credentials and set up a client. For Rackspace users, you will need your username and API key:

```
use OpenCloud\Rackspace;

$client = new Rackspace(Rackspace::US_IDENTITY_ENDPOINT, array(
  'username' => '{username}',
  'apiKey'   => '{apiKey}',
));
```

### OpenStack setup

If you're an OpenStack user, you will also need to prove a few other configuration parameters:

```
$client = new OpenCloud\OpenStack('{keystoneUrl}', array(
  'username' => '{username}',
  'password' => '{apiKey}',
  'tenantId' => '{tenantId}',
));
```

### Databases service

Now to instantiate the Databases service:

```
$service = $client->databaseService('{catalogName}', '{region}', '{urlType}');
```

- `{catalogName}` is the name of the service as it appears in the service catalog. OpenStack users *must* set this value. For Rackspace users, a default will be provided if you pass in `null`.
- `{region}` is the region the service will operate in. For Rackspace users, you can select one of the following from the *supported regions page*.
- `{urlType}` is the *type of URL* to use, depending on which endpoints your catalog provides. If omitted, it will default to the public network.

## Operations

### Instances

### Create a new instance

```
// Create an empty object
$instance = $service->instance();

// Send to the API
$instance->create(array(
    'name'   => '{name}',
    'flavor' => $service->flavor('{flavorId}'),
    'volume' => array('size' => 4)               // 4GB of volume disk
));
```

Get the executable PHP script for this sample

### Waiting for the instance to build

The SDK provides a blocking operation that will wait until your instance resource has transitioned into an `ACTIVE` state. During this period, it will continuously poll the API and break the loop when the state has been achieved:

```
$instance->waitFor('ACTIVE', null, function ($instance) {
    // This will be executed continuously
    printf("Database instance build status: %s\n", $instance->status);
});
```

### Connecting an instance to a load balancer

The instance created in the previous step can only be accessed from the Rackspace private network (aka SERVICENET). If you have a cloud server instance in the same region as the database server instance, you will be able to connect to the database from that cloud server instance.

If, however, you would like to access the database from the Internet, you will need to create a load balancer with an IP address that is routable from the Internet and attach the database server instance as a back-end node of this load balancer.

```
$lbService = $client->loadBalancerService(null, '{region}');

// Create empty object
$loadBalancer = $lbService->loadBalancer();

// Associate this LB with the instance as a "node"
$loadBalancer->addNode($instance->hostname, 3306);
$loadBalancer->addVirtualIp('PUBLIC');

// Configure other parameters and send to the API
$loadBalancer->create(array(
  'name'     => 'DB Load Balancer',
  'port'     => 3306,
  'protocol' => 'MYSQL',
));

// Wait for the resource to create
$loadBalancer->waitFor('ACTIVE', null, function ($loadBalancer) {
    printf("Load balancer build status: %s\n", $loadBalancer->status);
});

foreach ($loadBalancer->virtualIps as $vip) {
    if ($vip->type == 'PUBLIC') {
        printf("Load balancer public %s address: %s\n", $vip->ipVersion, $vip->
↪address);
    }
}
```

In the example above, a load balancer is created with the database server instance as its only back-end node. Further, this load balancer is configured to listen for MySQL connections on port 3306. Finally a virtual IP address (VIP) is configured in the PUBLIC network address space so that this load balancer may receive connections from the Internet.

Once the load balancer is created and becomes ACTIVE, it's Internet-accessible IP addresses are printed out. If you connect to any of these IP addresses on port 3306 using the MySQL protocol, you will be connected to the database created in step 3.

### Retrieving an instance

```
$instance = $service->instance('{instanceId}');
```

Get the executable PHP script for this example

### Updating an instance

An instance can be updated to use a specific configuration as shown below.

```
$instance->update(array(
    'configuration' => '{configurationId}'
));
```

**Note:** If any parameters in the associated configuration require a restart, then you will need to *restart the instance* after the update.

### Deleting an instance

```
$instance->delete();
```

### Restarting an instance

```
$instance->restart();
```

### Resizing an instance's RAM

To change the amount of RAM allocated to the instance:

```
$flavor = $service->flavor('{flavorId}');
$instance->resize($flavor);
```

### Resizing an instance's volume

You can also independently change the volume size to increase the disk space:

```
// Increase to 8GB disk
$instance->resizeVolume(8);
```

### Databases

### Setup

In order to interact with the functionality of databases, you must first retrieve the details of the instance itself. To do this, you must substitute *{instanceId}* for your instance's ID:

```
$instance = $service->instance('{instanceId}');
```

### Creating a new database

To create a new database, you must supply it with a name; you can optionally specify its character set and collating sequence:

```
// Create an empty object
$database = $instance->database();

// Send to API
$database->create(array(
    'name'          => 'production',
    'character_set' => 'utf8',
    'collate'       => 'utf8_general_ci'
));
```

You can find values for `character_set` and `collate` at the MySQL website.

### Deleting a database

```
$database->delete();
```

**Note:** This is a destructive operation: all your data will be wiped away and will not be retrievable.

### Listing databases

```
$databases = $service->databaseList();

foreach ($databases as $database) {
    /** @param $database OpenCloud\Database\Resource\Database */
}
```

### Users

### Setup

Finally, in order to interact with the functionality of databases, you must first retrieve the details of the instance itself. To do this, you must substitute *{instanceId}* for your instance's ID:

```
$instance = $service->instance('{instanceId}');
```

### Creating users

Database users exist at the `Instance` level, but can be associated with a specific `Database`. They are represented by the `OpenCloud\Database\Resource\User` class.

```
// New instance of OpenCloud\Database\Resource\User
$user = $instance->user();

// Send to API
$user->create(array(
    'name'      => 'Alice',
    'password'  => 'fooBar'
```

```
    'databases' => array('production')
));
```

### Deleting a user

```
$user->delete();
```

### The root user

By default, Cloud Databases does not enable the root user. In most cases, the root user is not needed, and having one can leave you open to security violations. However, if you do want to enable access to the root user:

```
$rootUser = $instance->enableRootUser();
```

This returns a regular `User` object with the `name` attribute set to `root` and the `password` attribute set to an auto-generated password.

### Check if root user is enabled

```
// true for yes, false for no
$instance->isRootEnabled();
```

### Grant database access

To grant access to one or more databases, you can run:

```
$user = $instance->user('{userName}');
$user->grantDbAccess(['{dbName1}', '{dbName2}']);
```

### Datastores

### Listing datastores

You can list out all the datastores available as shown below:

```
$datastores = $service->datastoreList();
foreach ($datastores as $datastore) {
    /** @var $datastore OpenCloud\Database\Resource\Datastore **/
}
```

Get the executable PHP script for this example

### Retrieving a datastore

You can retrieve a specific datastore's information, using its ID, as shown below:

```
/** @var OpenCloud\Database\Resource\Datastore **/
$datastore = $service->datastore('{datastoreId}');
```

Get the executable PHP script for this example

### Listing datastore versions

You can list out all the versions available for a specific datastore, as shown below:

```
$versions = $datastore->versionList();
foreach ($versions as $version) {
    /** @var $version OpenCloud\Database\Resource\DatastoreVersion **/
}
```

Get the executable PHP script for this example

### Retrieving a datastore version

You a retrieve a specific datastore version, using its ID, as shown below:

```
$datastoreVersion = $datastore->version('{versionId}');
```

Get the executable PHP script for this example

## Glossary

**configuration group** A configuration group is a collection of key/value pairs which configure a database instance. Some directives are capable of being applied dynamically, while other directives require a server restart to take effect. The configuration group can be applied to an instance at creation or applied to an existing instance to modify the behavior of the running datastore on the instance.

**flavor** A flavor is an available hardware configuration for a database instance. Each flavor has a unique combination of memory capacity and priority for CPU time.

**instance** A database instance is an isolated MySQL instance in a single tenant environment on a shared physical host machine. Also referred to as instance.

**database** A database is a local MySQL database running on an instance.

**user** A user is a local MySQL user that can access a database running on an instance.

**datastore** The database engine running on your instance. Currently, there is support for MySQL 5.6, MySQL 5.1, Percona 5.6 and MariaDB 10.

**volume** A volume is user-specified storage that contains the database engine data directory. Volumes are automatically provisioned on shared Internet Small Computer System Interface (iSCSI) storage area networks (SAN) that provide for increased performance, scalability, availability and manageability. Applications with high I/O demands are performance optimized and data is protected through both local and network RAID-10.

## Further Links

- Getting Started Guide for the API
- API Developer Guide

- API release history

# DNS v1

**Note:** This service is only available for Rackspace users.

## Setup

The first step is to pass in your credentials and set up a client. For Rackspace users, you will need your username and API key:

```
use OpenCloud\Rackspace;

$client = new Rackspace(Rackspace::US_IDENTITY_ENDPOINT, array(
  'username' => '{username}',
  'apiKey'   => '{apiKey}',
));
```

### DNS service

Now to instantiate the DNS service:

```
$service = $client->dnsService();
```

## Operations

### Records

### Setup

In order to interact with the functionality of records, you must first retrieve the details of the domain itself. To do this, you must substitute *{domainId}* for your domain's ID:

```
$domain = $service->domain('{domainId}');
```

### Get record

In order to retrieve details for a specific DNS record, you will need its **id**:

```
$record = $domain->record('{recordId}');
```

If you do not have this ID at your disposal, you can traverse the record collection and do a string comparison (detailed below).

### List records

This call lists all records configured for the specified domain.

```
$records = $domain->recordList();

foreach ($records as $record) {
    printf("Record name: %s, ID: %s, TTL: %s\n", $record->name, $record->id, $record->
↪ttl);
}
```

### Query parameters

You can pass in an array of query parameters for greater control over your search:

| Name | Data type | Description |
|---|---|---|
| `type` | `string` | The record type |
| `name` | `string` | The record name |
| `data` | `string` | Data for this record |

### Find a record ID from its name

For example:

```
$records = $domain->recordList(array(
    'name' => 'imap.example.com',
    'type' => 'MX'
));

foreach ($records as $record) {
    $recordId = $record->id;
}
```

### Add record

This call adds a new record to the specified domain:

```
$record = $domain->record(array(
    'type' => 'A',
    'name' => 'example.com',
    'data' => '192.0.2.17',
    'ttl'  => 3600
));

$record->create();
```

Please be aware that records that are added with a different hostname than the parent domain might fail silently.

### Modify record

```
$record = $domain->record('{recordId}');
$record->ttl -= 100;
$record->update();
```

### Delete record

```
$record->delete();
```

## Domains

### Get domain

To retrieve a specific domain, you will need the domain's **id**, not its domain name:

```
$domain = $service->domain('{domainId}');
```

If you are having trouble remembering or accessing the domain ID, you can do a domain list search for your domain and then access its ID.

### List domains

These calls provide a list of all DNS domains manageable by a given account. The resulting list is flat, and does not break the domains down hierarchically by subdomain. All representative domains are included in the list, even if a domain is conceptually a subdomain of another domain in the list.

```
$domains = $service->domainList();

# Return detailed information for each domain
$domains = $service->domainList(true);
```

### Filter parameters

You can filter the search by using the `name` parameter in a key/value array supplied as a method argument. For example, to retrieve domains named `foo.com`, along with any subdomains like `bar.foo.com`:

```
$hoolaDomains = $service->domainList(array(
    'name' => 'foo.com'
));
```

Filter criteria may consist of:

- Any letter (A-Za-z)
- Numbers (0-9)
- Hyphen ("-")
- 1 to 63 characters

Filter criteria should not include any of the following characters:

‘ + , | ! ” £ $ % & / ( ) = ? ^ * ç ° § ; : _ > ] [ @ à, é, ò

### Finding a domain ID

Once you have a list of domains, to retrieve a domain's ID:

```
foreach ($domains as $domain) {
    $id = $domain->id;
}
```

### List domain changes

This call shows all changes to the specified domain since the specified date/time. To list all available changes for a domain for the current day:

```
$changes = $domain->changes();
```

For more granular control, you can manually define the `since` parameter like so:

```
$since = date('c', strtotime('last week'));
$changes = $domain->changes($since);
```

Once you have a set of changes, you can iterate over them like so:

```
foreach ($changes->changes as $change) {
    printf("Domain: %s\nAction: %s\nTarget: %s", $change->domain, $change->action,
↪$change->targetType);

    foreach ($change->changeDetails as $detail) {
        printf("Details: %s was changed from %s to %s", $detail->field, $detail->
↪oldValue, $detail->newValue);
    }
}
```

### Create domain

The first thing you will need to do is instantiate a new object and set the primary `A` record for the DNS domain, like so:

```
// get empty object
$domain = $service->domain();

// add A record
$aRecord = $domain->record(array(
    'type' => 'A',
    'name' => 'example.com',
    'data' => '192.0.2.17',
    'ttl'  => 3600
));

$domain->addRecord($aRecord);
```

You also have the option of adding more types of DNS records such as `CNAME`, `MX` and `NS` records. This step is completely optional and depends on your requirements:

```
// add CNAME record
$cRecord = $domain->record(array(
    'type' => 'CNAME',
    'name' => 'www.example.com',
    'data' => 'example.com',
    'ttl'  => 3600
));
$domain->addRecord($cRecord);

// add MX record
$mxRecord = $domain->record(array(
    'type' => 'MX',
    'data' => 'mail.example.com',
    'name' => 'example.com',
    'ttl'  => 3600,
    'priority' => 5
));
$domain->addRecord($mxRecord);

// add NS record
$nsRecord = $domain->record(array(
    'type' => 'NS',
    'data' => 'dns1.stabletransit.com',
    'name' => 'example.com',
    'ttl'  => 5400
));
$domain->addRecord($nsRecord);
```

You can also add sub-domains to your new DNS domain. Again, this is completely optional:

```
$subdomain = $domain->subdomain(array(
    'emailAddress' => 'foo@example.com',
    'name'         => 'dev.example.com',
    'comment'      => 'Dev portal'
));
$domain->addSubdomain($subdomain);
```

Once you've finished configuring how your DNS domain will work, you're ready to specify the essential details and send it to the API for creation:

```
$domain->create(array(
    'emailAddress' => 'webmaster@example.com',
    'ttl'          => 3600,
    'name'         => 'example.com',
    'comment'      => 'Optional comment'
));
```

### Clone domain

This call will duplicate an existing domain under a new name. By default, all records and, optionally, subdomains are duplicated as well.

The method signature you will need to use is:

**cloneDomain** (*$newDomainName*[, *$subdomains*[, *$comments*[, *$email*[, *$records* ] ] ] ])
Clone a domain

**Parameters**

- **$newDomainName** (*string*) – The name of the new domain entry
- **$subdomains** (*bool*) – Set to `true` to clone all the subdomains for this domain
- **$comments** (*bool*) – Set to `true` to replace occurrences of the reference domain name with the new domain name in comments on the cloned (new) domain.
- **$email** (*bool*) – Set to `true` to replace occurrences of the reference domain name with the new domain name in data fields (of records) on the cloned (new) domain. Does not affect NS records.
- **$records** (*bool*) – Set to `true` to replace occurrences of the reference domain name with the new domain name in data fields (of records) on the cloned (new) domain. Does not affect NS records.

For example:

```
$asyncResponse = $domain->cloneDomain('new-name.com', true, false, true, false);
```

### Export domain

This call provides access to the BIND (Berkeley Internet Name Domain) 9 for the requested domain. This call is for a single domain only, and as such, does not traverse up or down the domain hierarchy for details:

```
$asyncResponse = $domain->export();

$body = $asyncResponse->waitFor('COMPLETED');
echo $body['contents'];
```

### Import domain

This operation will create a new DNS domain according to a BIND (Berkeley Internet Name Domain) 9 formatted value.

In order for the BIND value to be considered valid, it needs to adhere to the following rules:

- Each record starts on a new line and on the first column. If a record will not fit on one line, use the BIND_9 line continuation convention where you put a left parenthesis and continue the one record on the next line and put a right parenthesis when the record ends. For example:

  example2.net. 3600 IN SOA dns1.stabletransit.com. (sample@rackspace.com. 1308874739 3600 3600 3600 3600)

- The attribute values of a record must be separated by a single blank or tab. No other white space characters.
- If there are any NS records, the data field should not be `dns1.stabletransit.com` or `dns2.stabletransit.com`. They will result in "duplicate record" errors.

For example:

```
$bind9Data = <<<EOT

example.net. 3600 IN SOA dns1.stabletransit.com. sample@rackspace.com. 1308874739␣
↪3600 3600 3600 3600
```

```
example.net. 86400 IN A 110.11.12.16
example.net. 3600 IN MX 5 mail2.example.net.
www.example.net. 5400 IN CNAME example.net.

EOT;

$asyncResponse = $service->import($bind9Data);
```

### Modify domain

Only the TTL, email address and comment attributes of a domain can be modified. Records cannot be added, modified, or removed through this API operation - you will need to use the add records, modify records or remove records operations respectively.

```
$domain->update(array(
    'ttl'          => ($domain->ttl + 100),
    'emailAddress' => 'new_dev@foo.com'
));
```

### Delete domain

```
$domain->delete();
```

### Limits

### List all limits

This call provides a list of all applicable limits for the specified account.

```
$limits = $service->limits();
```

### Absolute limits

There are some absolute limits imposed on your account - such as how many domains you can create and how many records you can create for each domain:

```
$absoluteLimits = $limits->absolute;

// Domain limit
echo $absoluteLimits->domains;

// Record limit per domain
echo $absoluteLimits->{'records per domain'};
```

### List limit types

To find out the different limit types you can query, run:

```
$limitTypes = $service->limitTypes();
```

will return:

```
array(3) {
  [0] => string(10) "RATE_LIMIT"
  [1] => string(12) "DOMAIN_LIMIT"
  [2] => string(19) "DOMAIN_RECORD_LIMIT"
}
```

#### Query a specific limit

```
$limit = $service->limits('DOMAIN_LIMIT');
echo $limit->absolute->limits->value;
```

### Reverse DNS

#### Get PTR record

PTR records refer to a parent device: either a Cloud Server or a Cloud Load Balancer with a public virtual IP address. You must supply a fully formed resource object in order to retrieve either one's PTR record:

```
/** @param $parent OpenCloud\DNS\Resource\HasPtrRecordsInterface */

$ptr = $service->ptrRecord(array(
    'parent' => $parent
));
```

So, in the above example, the `$parent` object could be an instance of `OpenCloud\Compute\Resource\Server` or `OpenCloud\LoadBalancer\Resource\LoadBalancer` - because they both implement `OpenCloud\DNS\Resource\HadPtrRecordsInterface`. Please consult the server documentation and load balancer documentation for more detailed usage instructions.

#### List PTR records

```
/** @param $parent OpenCloud\DNS\Resource\HasPtrRecordsInterface */

$ptrRecords = $service->ptrRecordList($parent);

foreach ($ptrRecords as $ptrRecord) {

}
```

#### Add PTR record

```
$parent = $computeService->server('foo-server-id');

$ptr = $dnsService->ptrRecord(array(
    'parent' => $parent,
```

```
    'ttl'     => 3600,
    'name'    => 'example.com',
    'type'    => 'PTR',
    'data'    => '192.0.2.7'
));

$ptr->create();
```

Here is a table that explains the above attributes:

| Name | Description | Required |
|---|---|---|
| type | Specifies the record type as "PTR". | Yes |
| name | Specifies the name for the domain or subdomain. Must be a valid domain name. | Yes |
| data | The data field for PTR records must be a valid IPv4 or IPv6 IP address. | Yes |
| ttl | If specified, must be greater than 300. Defaults to 3600 if no TTL is specified. | No |
| comment | If included, its length must be less than or equal to 160 characters. | No |

### Modify PTR record

```
$ptr->update(array(
    'ttl' => $ptr->ttl * 2
));
```

### Delete PTR record

```
$ptr->delete();
```

## Glossary

**domain** A domain is an entity/container of all DNS-related information containing one or more records.

**record** A DNS record belongs to a particular domain and is used to specify information about the domain. There are several types of DNS records. Each record type contains particular information used to describe that record's purpose. Examples include mail exchange (MX) records, which specify the mail server for a particular domain, and name server (NS) records, which specify the authoritative name servers for a domain.

**subdomain** Subdomains are domains within a parent domain, and subdomains cannot be registered. Subdomains allow you to delegate domains. Subdomains can themselves have subdomains, so third-level, fourth-level, fifth-level, and deeper levels of nesting are possible.

**pointer records** DNS usually determines an IP address associated with a domain name. Reverse DNS is the opposite process: resolving a domain name from an IP address. This is usually achieved with a domain name pointer.

## Further Links

- Getting Started Guide for the API
- API Developer Guide
- API release history

# Identity v2

## Setup

### Rackspace setup

The first step is to pass in your credentials and set up a client. For Rackspace users, you will need your username and API key:

```
use OpenCloud\Rackspace;

$client = new Rackspace(Rackspace::US_IDENTITY_ENDPOINT, array(
  'username' => '{username}',
  'apiKey'   => '{apiKey}',
));
```

### OpenStack setup

If you're an OpenStack user, you will also need to prove a few other configuration parameters:

```
$client = new OpenCloud\OpenStack('{keystoneUrl}', array(
  'username' => '{username}',
  'password' => '{apiKey}',
  'tenantId' => '{tenantId}',
));
```

### Identity service

Now to instantiate the Identity service:

```
$service = $client->identityService();
```

## Operations

### Tokens

### Create token (authenticate)

In order to generate a token, you must pass in the JSON template that is sent to the API. This is because Rackspace's operation expects a slightly different entity body than OpenStack Keystone.

To do this, and then generate a token:

```
$json = $client->getCredentials();

/** @var $response Guzzle\Http\Message\Response */
$response = $service->generateToken($json);
$jsonBody = $response->json();
```

When a token is generated by the API, there are a few things returned:

- a service catalog outlining all of the services you can interact with, including their names, service types, and endpoint URLs. Which services make up your catalog, and how your catalog is structured, will depend on your service provider.
- details about your token, such as its ID, created and expiration date
- details about your user account
- details about your tenant

### Interacting with the service catalog

Once you have the `$jsonBody`, you can construct a `Catalog` object for easier interaction:

```
$data = $jsonBody->access->serviceCatalog;
$catalog = OpenCloud\Common\Service\Catalog::factory($data);

foreach ($catalog->getItems() as $service) {
  /** @param $service OpenCloud\Common\Service\CatalogItem */
  printf("Catalog item: Name [%s] Type [%s]\n", $service->getName(), $service->
↪getType());

  foreach ($service->getEndpoints() as $endpoint) {
    printf("  Endpoint provided: Region [%s] PublicURL [%s] PrivateURL [%s]\n",
      $endpoint->getRegion(), $endpoint->getPublicUrl(), $endpoint->getPrivateUrl());
  }
}
```

### Interacting with tokens

```
$data = $jsonBody->access->token;
$token = $service->resource('Token', $data);

printf("Token ID: %s - Token expiry %s", $token->getId(), $token->getExpires());

if ($token->hasExpired()) {
  // ...
}
```

### Interacting with users

```
$data = $jsonBody->access->user;
$user = $service->resource('User', $data);
```

To see which methods you can call on `$user` (which implements `OpenCloud\Identity\Resource\User`), see our *user documentation* which accompanies this guide.

### Interacting with tenants

```
$data = $jsonBody->access->tenant;
$tenant = $service->resource('Tenant', $data);
```

To see which methods you can call on `$tenant` (which implements `OpenCloud\Identity\Resource\Tenant`), see our user documentation which accompanies this guide.

### Revoke token (destroy session)

```
$service->revokeToken('{tokenId}');
```

### Users

### Object properties/methods

| Property | Description | Getter | Setter |
|---|---|---|---|
| id | The unique ID for this user | `getId()` | `setId()` |
| username | Username for this user | `getUsername()` | `setUsername()` |
| email | User's email address | `getEmail()` | `setEmail()` |
| enabled | Whether or not this user can consume API functionality | `getEnabled()` or `isEnabled()` | `setEnabled()` |
| password | Either a user-defined string, or an automatically generated one, that provides security when authenticating. | `getPassword()` only valid on creation | `setPassword()` to set local property only. To set password on API (retention), use `updatePassword()`. |
| defaultRegion | Default region associates a user with a specific regional datacenter. If a default region has been assigned for this user and that user has **NOT** explicitly specified a region when creating a service object, the user will obtain the service from the default region. | `getDefaultRegion()` | `setDefaultRegion()` |
| domainId | Domain ID associates a user with a specific domain which was assigned when the user was created or updated. A domain establishes an administrative boundary for a customer and a container for a customer's tenants (accounts) and users. Generally, a domainId is the same as the primary tenant id of your cloud account. | `getDomainId()` | `setDomainId()` |

### List users

```
$users = $service->getUsers();

foreach ($users as $user) {
    // ...
}
```

Get the executable PHP script for this example

### Retrieve a user by username

```
$user = $service->getUser('jamie');
```

Get the executable PHP script for this example

### Retrieve a user by user ID

```
use OpenCloud\Identity\Constants\User as UserConst;

$user = $service->getUser('{userId}', UserConst::MODE_ID);
```

Get the executable PHP script for this example

### Retrieve a user by email address

```
use OpenCloud\Identity\Constants\User as UserConst;

$user = $service->getUser('{emailAddress}', UserConst::MODE_EMAIL);
```

Get the executable PHP script for this example

### Create user

There are a few things to bear in mind when creating a user:

- This operation is available only to users who hold the `identity:user-admin` role. This admin can create a user who holds the `identity:default` user role.
- The created user **will** have access to APIs but **will not** have access to the Cloud Control Panel.
- A maximum of 100 account users can be added per account.
- If you attempt to add a user who already exists, an HTTP error 409 results.

The `username` and `email` properties are required for creating a user. Providing a `password` is optional; if omitted, one will be automatically generated and provided in the response.

```
use Guzzle\Http\Exception\ClientErrorResponseException;

 $user = $service->createUser(array(
    'username' => 'newUser',
    'email'    => 'foo@bar.com'
 ));

// show generated password
echo $user->getPassword();
```

Get the executable PHP script for this example

### Update user

When updating a user, specify which attribute/property you want to update:

```
$user->update(array(
    'email' => 'new_email@bar.com'
));
```

### Updating a user password

Updating a user password requires calling a distinct method:

```
$user->updatePassword('password123');
```

### Delete user

```
$user->delete();
```

Get the executable PHP script for this example

### List credentials

This operation allows you to see your non-password credential types for all authentication methods available.

```
$creds = $user->getOtherCredentials();
```

### Get user API key

```
echo $user->getApiKey();
```

### Reset user API key

When resetting an API key, a new one will be automatically generated for you:

```
$user->resetApiKey();
echo $user->getApiKey();
```

Get the executable PHP script for this example

### Tenants

### List tenants

```
$tenants = $service->getTenants();

foreach ($tenants as $tenant) {
    // ...
}
```

#### Tenant object properties and methods

Once you have a `OpenCloud\Identity\Resource\Tenant` object, you can retrieve information like so:

```
$tenant->getId();
$tenant->getName();
$tenant->getDescription();
$tenant->isEnabled();
```

### Glossary

**token** A token is an opaque string that represents an authorization to access cloud resources. Tokens may be revoked at any time and are valid for a finite duration.

**tenant** A tenant is a container used to group or isolate resources and/or identity objects. Depending on the service operator, a tenant may map to a customer, account, organization, or project.

**user** A user is a digital representation of a person, system, or service who consumes cloud services. Users have credentials and may be assigned tokens; based on these credentials and tokens, the authentication service validates that incoming requests are being made by the user who claims to be making the request, and that the user has the right to access the requested resources. Users may be directly assigned to a particular tenant and behave as if they are contained within that tenant.

### Further Links

- Quickstart for the API
- API Developer Guide

## Images v1

### Setup

#### Rackspace setup

The first step is to pass in your credentials and set up a client. For Rackspace users, you will need your username and API key:

```
use OpenCloud\Rackspace;

$client = new Rackspace(Rackspace::US_IDENTITY_ENDPOINT, array(
  'username' => '{username}',
  'apiKey'   => '{apiKey}',
));
```

#### OpenStack setup

If you're an OpenStack user, you will also need to prove a few other configuration parameters:

```
$client = new OpenCloud\OpenStack('{keystoneUrl}', array(
  'username' => '{username}',
  'password' => '{apiKey}',
  'tenantId' => '{tenantId}',
));
```

### Images service

Now to instantiate the Images service:

```
$service = $client->imageService(null, '{region}');
```

## Operations

### Images

### List images

```
$images = $service->listImages();

foreach ($images as $image) {
   /** @param $image OpenCloud\Image\Resource\Image */
}
```

### Get image details

```
/** @param $image OpenCloud\Image\Resource\Image */
$image = $service->getImage('{imageId}');
```

### A note on schema classes

Both `OpenCloud\Image\Resource\Image` and `OpenCloud\Image\Resource\Member` extend the `AbstractSchemaResource` class, which offers some unique functionality.

Because these resources are inherently dynamic - i.e. they are modelled on dynamic JSON schema - you need to access their state in a different way than conventional getter/setter methods, and even class properties. For this reason, they implement SPL's native ArrayAccess interface which allows you to access their state as a conventional array:

```
$image = $service->getImage('{imageId}');

$id = $image['id'];
$tags = $image['tags'];
```

### Update image

You can only update your own custom images - you cannot update or delete base images. The way in which you may update your image is dictated by its schema.

Although you should be able to add new and replace existing properties, always prepare yourself for a situation where it might be forbidden:

```
use OpenCloud\Common\Exceptions\ForbiddenOperationException;

try {
    $image->update(array(
        'name'        => 'foo',
        'newProperty' => 'bar'
    ));
} catch (ForbiddenOperationException $e) {
    // A 403 Forbidden was returned
}
```

There are three operations that can take place for each Image property:

- If a `false` or `null` value is provided, a `REMOVE` operation will occur, removing the property from the JSON document
- If a non-false value is provided and the property does not exist, an `ADD` operation will add it to the document
- If a non-false value is provided and the property does exist, a `REPLACE` operation will modify the property in the document

### Delete image

```
$image->delete();
```

### JSON schemas

### Schema types

There are currently four types of schema: Images schema, Image schema, Members schema, and Member schema.

### Example response from the API

A sample response from the API, for an Images schema might be:

```
{
    "name": "images",
    "properties": {
        "images": {
            "items": {
                "type": "array",
                "name": "image",
                "properties": {
                    "id": {"type": "string"},
                    "name": {"type": "string"},
                    "visibility": {"enum": ["public", "private"]},
                    "status": {"type": "string"},
                    "protected": {"type": "boolean"},
                    "tags": {
                        "type": "array",
                        "items": {"type": "string"}
```

```
                    },
                    "checksum": {"type": "string"},
                    "size": {"type": "integer"},
                    "created_at": {"type": "string"},
                    "updated_at": {"type": "string"},
                    "file": {"type": "string"},
                    "self": {"type": "string"},
                    "schema": {"type": "string"}
                },
                "additionalProperties": {"type": "string"},
                "links": [
                    {"href": "{self}", "rel": "self"},
                    {"href": "{file}", "rel": "enclosure"},
                    {"href": "{schema}", "rel": "describedby"}
                ]
            }
        },
        "schema": {"type": "string"},
        "next": {"type": "string"},
        "first": {"type": "string"}
    },
    "links": [
        {"href": "{first}", "rel": "first"},
        {"href": "{next}", "rel": "next"},
        {"href": "{schema}", "rel": "describedby"}
    ]
}
```

The top-level schema is called `images`, and contains an array of links and a properties object. Inside this properties object we see the structure of this top-level `images` object. So we know that it will take this form:

```
{
   "images": [something...]
}
```

Within this object, we can see that it contains an array of anonymous objects, each of which is called `image` and has its own set of nested properties:

```
{
    "images": [
        {
            [object 1...]
        },
        {
            [object 2...]
        },
        {
            [object 3...]
        }
    ]
}
```

The structure of these nested objects are defined as another schema - i.e. a *subschema*. We know that each object has an ID property (string), a name property (string), a visibility property (can either be `private` or `public`), etc.

```
{
    "images": [
        {
```

```
            "id": "foo",
            "name": "bar",
            "visibility": "private",
            // etc.
        },
        {
            "id": "foo",
            "name": "bar",
            "visibility": "private",
            // etc.
        },
        {
            "id": "foo",
            "name": "bar",
            "visibility": "private",
            // etc.
        }
    ]
}
```

Each nested property of a schema is represented by the `OpenCloud\Image\Resource\Schema\Property` class.

If you would like to find out more about schemas, Guzzle has good documentation about service descriptions, which is fairly analogous.

### JSON Patch

The Glance API has a unique way of updating certain dynamic resources: they use JSON Patch method, as outlined in RFC 6902.

Requests need to use the `application/openstack-images-v2.1-json-patch` content-type.

In order for the operation to occur, the request entity body needs to contain a very particular structure:

```
[
    {"op": "replace", "path": "/name", "value": "Fedora 17"},
    {"op": "replace", "path": "/tags", "value": ["fedora", "beefy"]}
]
```

- The `op` key refers to the type of Operation (see `OpenCloud\Image\Enum\OperationType` for a full list).
- The `path` key is a JSON pointer to the document property you want to modify or insert. JSON pointers are defined in RFC 6901.
- The `value` key is the value.

Because this is all handled for you behind the scenes, we will not go into exhaustive depth about how this operation is handled. You can browse the source code, consult the various RFCs and the official documentation for additional information.

### Sharing images

Images can be created and deleted by image producers, updated by image consumers, and listed by both image producers and image consumers:

| Operation | Producer can? | Consumer can? |
|---|---|---|
| Created | Yes | No |
| Deleted | Yes | No |
| Updated | No | Yes |
| Listed | Yes | Yes |

The producer shares an image with the consumer by making the consumer a *member* of that image. The consumer then accepts or rejects the image by changing the member status. Once accepted, the image appears in the consumer's image list.

### Typical workflow

1. The producer posts the availability of specific images on a public website.
2. A potential consumer provides the producer with his/her tenant ID and email address.
3. The producer **'creates a new Image Member <>'__** with the consumer's details
4. The producer notifies the consumer via email that the image has been shared and provides the image's ID.
5. If the consumer wishes the image to appear in his/her image list, the consumer **'updates their own Member status <>'__** to `ACCEPTED`.

### Additional notes

- If the consumer subsequently wishes to hide the image, the consumer can change their Member status to `REJECTED`.
- If the consumer wishes to hide the image, but is open to the possibility of being reminded by the producer that the image is available, the consumer can change their Member status to `PENDING`.
- Image producers add or remove image members, but may not modify the member status of an image member.
- Image consumers change their own member status, but may not add or remove themselves as an image member.
- Image consumers can boot from any image shared by the image producer, regardless of the member status, as long as the image consumer knows the image ID.

### Setup

All member operations are executed against an Image, so you will need to set one up first:

```
$image = $service->getImage('{imageId}');
```

### List image members

This operation is available for both producers and consumers.

```
$members = $image->listMembers();

foreach ($members as $member) {
    /** @param $member OpenCloud\Image\Resource\Member */
}
```

### Create image member

This operation is only available for producers.

```
/** @param $response Guzzle\Http\Message\Response */
$response = $image->createMember('{tenantId}');
```

### Delete image member

This operation is only available for producers.

```
/** @param $member OpenCloud\Image\Resource\Member */
$member = $image->getMember('{tenantId}');
$member->delete();
```

### Update image member status

This operation is only available for consumers.

```
use OpenCloud\Images\Enum\MemberStatus;

/** @param $member OpenCloud\Image\Resource\Member */
$member = $image->getMember('{tenantId}');

$member->updateStatus(MemberStatus::ACCEPTED);
```

The acceptable states you may pass in are made available to you through the constants defined in the `OpenCloud\Images\Enum\MemberStatus` class.

### Image tags

### Setup

All member operations are executed against an Image, so you will need to set one up first:

```
$image = $service->getImage('{imageId}');
```

### Add image tag

```
/** @param $response Guzzle\Http\Message\Response */
$response = $image->addTag('jamie_dev');
```

### Delete image tag

```
/** @param $response Guzzle\Http\Message\Response */
$response = $image->deleteTag('jamie_dev');
```

### Glossary

**image** A virtual machine image is a single file which contains a virtual disk that has an installed bootable operating system. In the Cloud Images API, an image is represented by a JSON-encoded data structure (the image schema) and its raw binary data (the image file).

**schema** The Cloud Images API supplies JSON documents describing the JSON-encoded data structures that represent domain objects, so that a client knows exactly what to expect in an API response.

**tag** An image tag is a string of characters used to identify a specific image or images.

### Further Links

- Getting Started Guide for the API
- API Developer Guide

## Load Balancer v1

**Note:** This service is only available for Rackspace users.

### Setup

The first step is to pass in your credentials and set up a client. For Rackspace users, you will need your username and API key:

```
use OpenCloud\Rackspace;

$client = new Rackspace(Rackspace::US_IDENTITY_ENDPOINT, array(
  'username' => '{username}',
  'apiKey'   => '{apiKey}',
));
```

#### Load Balancer service

Now to instantiate the Load Balancer service:

```
$service = $client->loadBalancerService('{catalogName}', '{region}', '{urlType}');
```

- `{catalogName}` is the name of the service as it appears in the service catalog. OpenStack users *must* set this value. For Rackspace users, a default will be provided if you pass in `null`.
- `{region}` is the region the service will operate in. For Rackspace users, you can select one of the following from the *supported regions page*.
- `{urlType}` is the *type of URL* to use, depending on which endpoints your catalog provides. If omitted, it will default to the public network.

## Operations

### Load Balancer

**Note:** Many of the examples in this document use two cloud servers as nodes for the load balancer. The variables `$serverOne` and `$serverTwo` refer to these two cloud servers.

#### Create Load Balancer

The first step is to instantiate an empty object, like so:

```
$loadBalancer = $service->loadBalancer();
```

In essence, all a load balancer does is evenly distribute traffic between various back-end nodes - which can be Compute or Database instances. So we will need to add a few when creating our load balancer:

```
$serverOneNode = $loadBalancer->node();
$serverOneNode->address = $serverOne->addresses->private[0]->addr;
$serverOneNode->port = 8080;
$serverOneNode->condition = 'ENABLED';

$serverTwoNode = $loadBalancer->node();
$serverTwoNode->address = $serverTwo->addresses->private[0]->addr;
$serverTwoNode->port = 8080;
$serverTwoNode->condition = 'ENABLED';
```

All that remains is apply final configuration touches, such as name and the port number, before submitting to the API:

```
$loadBalancer->addVirtualIp('PUBLIC');
$loadBalancer->create(array(
    'name'      => 'My load balancer',
    'port'      => 80,
    'protocol'  => 'HTTP',
    'nodes'     => array($serverOneNode, $serverTwoNode),
    'algorithm' => 'ROUND_ROBIN',
));
```

For a full list of available *protocols* and *algorithms* please see the sections below.

Get the executable PHP script for this example

#### Get Load Balancer Details

You can retrieve a single load balancer's details by using its ID:

```
/** @var $loadBalancer OpenCloud\LoadBalancer\Resource\LoadBalancer **/
$loadBalancer = $service->loadBalancer('{loadBalancerId}');
```

#### List Load Balancers

You can retrieve a list of all your load balancers:

```
$loadBalancers = $service->loadBalancerList();

foreach ($loadBalancers as $loadBalancer) {
    /** @var $loadBalancer OpenCloud\LoadBalancer\Resource\LoadBalancer **/
}
```

Get the executable PHP script for this example

### Update a Load Balancer

You can update one or more of the following load balancer attributes:

- `name`: The name of the load balancer
- `algorithm`: The algorithm used by the load balancer to distribute traffic amongst its nodes. See also: *Load balancing algorithms*.
- `protocol`: The network protocol used by traffic coming in to the load balancer. See also: *Protocols*.
- `port`: The network port on which the load balancer listens for incoming traffic.
- `halfClosed`: Enable or Disable Half-Closed support for the load balancer.
- `timeout`: The timeout value for the load balancer to communicate with its nodes.
- `httpsRedirect`: Enable or disable HTTP to HTTPS redirection for the load balancer. When enabled, any HTTP request will return status code 301 (Moved Permanently), and the requestor will be redirected to the requested URL via the HTTPS protocol on port 443. For example, http://example.com/page.html would be redirected to https:// example.com/page.html. Only available for HTTPS protocol (`port` = 443), or HTTP Protocol with a properly configured SSL Termination ('secureTrafficOnly=true, securePort=443). See also: *SSL Termination*.

```
$loadBalancer->update(array(
    'name'      => 'New name',
    'algorithm' => 'ROUND_ROBIN'
));
```

### Remove Load Balancer

When you no longer have a need for the load balancer, you can remove it:

```
$loadBalancer->delete();
```

Get the executable PHP script for this example

### Protocols

When a load balancer is created a network protocol must be specified. This network protocol should be based on the network protocol of the back-end service being load balanced. Common protocols are `HTTP`, `HTTPS` and `MYSQL`. A full list is available here.

### List Load Balancing Protocols

You can list all supported network protocols like so:

```
$protocols = $service->protocolList();

foreach ($protocols as $protocol) {
    /** @var $protocol OpenCloud\LoadBalancer\Resource\Protocol **/
}
```

### Algorithms

Load balancers use an **algorithm** to determine how incoming traffic is distributed amongst the back-end nodes. A full list is available here.

### List Load Balancing Algorithms

You can programmatically list all supported load balancing algorithms:

```
$algorithms = $service->algorithmList();

foreach ($algorithms as $algorithm) {
    /** @var $algorithm OpenCloud\LoadBalancer\Resource\Algorithm **/
}
```

### Nodes

### Setup

In order to interact with this feature you must first retrieve a particular load balancer, like so:

```
$loadBalancer = $service->loadBalancer('{id}');
```

### List Nodes

You can list the nodes attached to a load balancer:

```
$nodes = $loadBalancer->nodeList();

foreach ($nodes as $node) {
    /** @var $node OpenCloud\LoadBalancer\Resource\Node **/
}
```

### Add Nodes

You can attach additional nodes to a load balancer. Assume `$loadBalancer` already has two nodes attached to it - `$serverOne` and `$serverTwo` - and you want to attach a third node to it, say `$serverThree`, which provides a service on port 8080.

**Important:** Remember to call `$loadBalancer->addNodes()` after all the calls to `$loadBalancer->addNode()` as shown below.

```
$address = $serverThree->addresses->private[0]->addr;
$loadBalancer->addNode($address, 8080);
$loadBalancer->addNodes();
```

The signature for `addNodes` is as follows:

**addNodes** (*$address*, *$port*[, *$condition* = *'ENABLED'*[, *$type* = *null*[, *$weight* = *null*]]])

Add a node to a load balancer

**Parameters**

- **$address** (*string*) – the IP address of the node
- **$port** (*integer*) – the port number of the node
- **$condition** (*string*) – the initial condition of the code. Defaults to `ENABLED`
- **$type** (*string*) – either `PRIMARY` or `SECONDARY`
- **$weight** (*integer*) – the node weight (for round-robin algorithm)

The `addNode` method accepts three more optional parameters, in addition to the two shown above:

### Modify Nodes

You can modify one or more of the following node attributes:

- `condition`: The condition of the load balancer:
  - `ENABLED` – Node is ready to receive traffic from the load balancer.
  - `DISABLED` – Node should not receive traffic from the load balancer.
  - `DRAINING` – Node should process any traffic it is already receiving but should not receive any further traffic from the load balancer.
- `type`: The type of the node:
  - `PRIMARY` – Nodes defined as PRIMARY are in the normal rotation to receive traffic from the load balancer.
  - `SECONDARY` – Nodes defined as SECONDARY are only in the rotation to receive traffic from the load balancer when all the primary nodes fail.
- `weight`: The weight, between 1 and 100, given to node when distributing traffic using either the `WEIGHTED_ROUND_ROBIN` or the `WEIGHTED_LEAST_CONNECTIONS` load balancing algorithm.

```
use OpenCloud\LoadBalancer\Enum\NodeCondition;
use OpenCloud\LoadBalancer\Enum\NodeType;

$node->update(array(
    'condition' => NodeCondition::DISABLED,
    'type'      => NodeType::SECONDARY
));
```

### Remove Nodes

There are two ways to remove a node. The first way is on an `OpenCloud\LoadBalancer\Resource\Node` instance, like so:

```
$node->delete();
```

The second is with an `OpenCloud\LoadBalancer\Resource\LoadBalancer` instance and the node's ID, like so:

```
$loadBalancer->removeNode('{nodeId}');
```

where '{nodeId}' is the integer ID of the node itself - this is a required value.

### View Node Service Events

You can view events associated with the activity between a node and a load balancer:

```
$nodeEvents = $loadBalancer->nodeEventList();

foreach ($nodeEvents as $nodeEvent) {
    /** @var $nodeEvent OpenCloud\LoadBalancer\Resource\NodeEvent **/
}
```

### Virtual IPs

### Setup

In order to interact with this feature you must first retrieve a particular load balancer, like so:

```
$loadBalancer = $service->loadBalancer('{id}');
```

### List Virtual IPs

You can list the VIPs associated with a load balancer like so:

```
$vips = $loadBalancer->virtualIpList();

foreach ($vips as $vip) {
    /** @var $vip of OpenCloud\LoadBalancer\Resource\VirtualIp **/
}
```

### Get existing VIP

To retrieve the details of an existing VIP on a load balancer, you will need its ID:

### Add Virtual IPv6

You can add additional IPv6 VIPs to a load balancer using the following method:

```
use OpenCloud\LoadBalancer\Enum\IpType;

$loadBalancer->addVirtualIp(IpType::PUBLIC, 6);
```

the first argument is the type of network your IP address will server traffic in - and can either be `PUBLIC` or `PRIVATE`. The second argument is the version of IP address, either `4` or `6`.

### Add Virtual IPv4

Similar to above:

```
use OpenCloud\LoadBalancer\Enum\IpType;

$loadBalancer->addVirtualIp(IpType::PUBLIC, 4);
```

### Remove Virtual IP

You can remove a VIP from a load balancer.

```
$vip->remove();
```

**Note:** A load balancer must have at least one VIP associated with it. If you try to remove a load balancer's last VIP, a `ClientErrorResponseException` will be thrown.

### Allowed Domains

### List Allowed Domains

You can list all allowed domains using a load balancer service object. An instance of `OpenCloud\Common\Collection\PaginatedIterator` is returned.

```
$allowedDomains = $service->allowedDomainList();

foreach ($allowedDomains as $allowedDomain) {
    /** @var $allowedDomain OpenCloud\LoadBalancer\Resource\AllowedDomain **/
}
```

### Access Lists

Access Lists allow fine-grained network access to a load balancer's VIP. Using access lists, network traffic to a load balancer's VIP can be allowed or denied from a single IP address, multiple IP addresses or entire network subnets.

Note that `ALLOW` network items will take precedence over `DENY` network items in an access list.

To reject traffic from all network items except those with the `ALLOW` type, add a `DENY` network item with the address of `0.0.0.0/0`.

### Setup

In order to interact with this feature you must first retrieve a particular load balancer, like so:

```
$loadBalancer = $service->loadBalancer('{id}');
```

### View Access List

You can view a load balancer's access list:

```
$accessList = $loadBalancer->accessList();

foreach ($accessList as $networkItem) {
    /** @var $networkItem OpenCloud\LoadBalancer\Resource\Access **/
}
```

### Add Network Items To Access List

You can add network items to a load balancer's access list very easily:

```
$loadBalancer->createAccessList(array(
    (object) array(
        'type'    => 'ALLOW',
        'address' => '206.160.165.1/24'
    ),
    (object) array(
        'type'    => 'DENY',
        'address' => '0.0.0.0/0'
    )
));
```

In the above example, we allowed access for 1 IP address, and used the "0.0.0.0" wildcard to blacklist all other traffic.

Get the executable PHP scripts for this example:

- Blacklist IP range
- Limit access to 1 IP

### Remove Network Item From Access List

You an remove a network item from a load balancer's access list:

```
$networkItem->delete();
```

### Content Caching

When content caching is enabled on a load balancer, recently-accessed files are stored on the load balancer for easy retrieval by web clients. Requests to the load balancer for these files are serviced by the load balancer itself, which reduces load off its back-end nodes and improves response times as well.

#### Setup

In order to interact with this feature you must first retrieve a particular load balancer, like so:

```
$loadBalancer = $service->loadBalancer('{id}');
```

#### Check Configuration

```
// TRUE if enabled, FALSE if not
$contentCaching = $loadBalancer->hasContentCaching();
```

#### Enable Content Caching

```
$loadBalancer->enableContentCaching(true);
```

#### Disable Content Caching

```
$loadBalancer->enableContentCaching(false);
```

### Error Pages

#### Setup

In order to interact with this feature you must first retrieve a particular load balancer, like so:

```
$loadBalancer = $service->loadBalancer('{id}');
```

An error page is the html file that is shown to the end user when an error in the service has been thrown. By default every virtual server is provided with the default error file. It is also possible to set a custom error page for a load balancer.

#### View Error Page Content

```
$errorPage = $loadBalancer->errorPage();
$errorPageContent = $errorPage->content;

/** @var $errorPageContent string **/
```

In the example above the value of `$errorPageContent` is the HTML for that page. This could either be the HTML of the default error page or of your custom error page.

#### Set Custom Error Page

```
$errorPage = $loadBalancer->errorPage();
$errorPage->update(array(
    'content' => '<HTML content of custom error page>'
));
```

#### Delete Custom Error Page

```
$errorPage = $loadBalancer->errorPage();
$errorPage->delete();
```

### Connection Logging

The connection logging feature allows logs to be delivered to a Cloud Files account every hour. For HTTP-based protocol traffic, these are Apache-style access logs. For all other traffic, this is connection and transfer logging.

#### Setup

In order to interact with this feature you must first retrieve a particular load balancer, like so:

```
$loadBalancer = $service->loadBalancer('{id}');
```

#### Check Configuration

```
// TRUE if enabled, FALSE if not
$connectionLogging = $loadBalancer->hasConnectionLogging();
```

#### Enable Connection Logging

```
$loadBalancer->enableConnectionLogging(true);
```

#### Disable Connection Logging

```
$loadBalancer->enableConnectionLogging(false);
```

### Health Monitors

#### Setup

In order to interact with this feature you must first retrieve a particular load balancer, like so:

```
$loadBalancer = $service->loadBalancer('{id}');
```

### Retrieve monitor details

```
/** @var $healthMonitor OpenCloud\LoadBalancer\Resource\HealthMonitor **/

$healthMonitor = $loadBalancer->healthMonitor();

printf(
    "Monitoring type: %s, delay: %s, timeout: %s, attempts before deactivation: %s",
    $healthMonitor->type, $healthMonitor->delay, $healthMonitor->timeout
);
```

For a full list, with explanations, of required and optional attributes, please consult the official documentation

### Update monitor

```
$healthMonitor->update(array(
    'delay'   => 120,
    'timeout' => 60,
    'type'    => 'CONNECT'
    'attemptsBeforeDeactivation' => 3
));
```

### Delete monitor

```
$healthMonitor->delete();
```

### Metadata

### Setup

In order to interact with this feature you must first retrieve a particular load balancer, like so:

```
$loadBalancer = $service->loadBalancer('{id}');
```

### List metadata

```
$metadataList = $loadBalancer->metadataList();

foreach ($metadataList as $metadataItem) {
    printf("Key: %s, Value: %s", $metadataItem->key, $metadataItem->value);
}
```

### Add metadata

```
$metadataItem = $loadBalancer->metadata();
$metadataItem->create(array(
    'key'   => 'foo',
```

```
    'value' => 'bar'
));
```

### Modify metadata

```
$metadataItem = $loadBalancer->metadata('foo');
$metadataItem->update(array(
    'value' => 'baz'
));
```

### Remove metadata

```
$metadataItem->delete();
```

### Session Persistence

There are two types (or modes) of session persistence:

| Name | Description |
|---|---|
| HTTP_COOKIE | A session persistence mechanism that inserts an HTTP cookie and is used to determine the destination back-end node. This is supported for HTTP load balancing only. |
| SOURCE_IP | A session persistence mechanism that will keep track of the source IP address that is mapped and is able to determine the destination back-end node. This is supported for HTTPS pass-through and non-HTTP load balancing only. |

### Setup

In order to interact with this feature you must first retrieve a particular load balancer, like so:

```
$loadBalancer = $service->loadBalancer('{id}');
```

### List Session Persistence Configuration

```
$sessionPersistence = $loadBalancer->sessionPersistence();

/** @var $sessionPersistenceType null | 'HTTP_COOKIE' | 'SOURCE_IP' **/
$sessionPersistenceType = $sessionPersistence->persistenceType;
```

In the example above:

- If session persistence is enabled, the value of `$sessionPersistenceType` is the type of session persistence: either `HTTP_COOKIE` or `SOURCE_IP`.
- If session persistence is disabled, the value of `$sessionPersistenceType` is `null`.

### Enable Session Persistence

```
$sessionPersistence = $loadBalancer->sessionPersistence();
$sessionPersistence->update(array(
    'persistenceType' => 'HTTP_COOKIE'
));
```

### Disable Session Persistence

```
$sessionPersistence = $loadBalancer->sessionPersistence();
$sessionPersistence->delete();
```

### SSL Termination

The SSL Termination feature allows a load balancer user to terminate SSL traffic at the load balancer layer versus at the web server layer. A user may choose to configure SSL Termination using a key and an SSL certificate or an (Intermediate) SSL certificate.

When SSL Termination is configured on a load balancer, a secure shadow server is created that listens only for secure traffic on a user-specified port. This shadow server is only visible to and manageable by the system. Existing or updated attributes on a load balancer with SSL Termination will also apply to its shadow server. For example, if Connection Logging is enabled on an SSL load balancer, it will also be enabled on the shadow server and Cloud Files logs will contain log files for both.

### Setup

In order to interact with this feature you must first retrieve a particular load balancer, like so:

```
$loadBalancer = $service->loadBalancer('{id}');
```

### View configuration

```
/** @var $sslConfig OpenCloud\LoadBalancer\Resource\SSLTermination **/
$sslConfig = $loadBalancer->SSLTermination();
```

### Update configuration

```
$sslConfig->update(array(
    'enabled'     => true,
    'securePort'  => 443,
    'privateKey'  => $key,
    'certificate' => $cert
));
```

For a full list, with explanations, of required and optional attributes, please consult the official documentation

Get the executable PHP script for this example

### Delete configuration

```
$sslConfig->delete();
```

## Statistics and Usage Reports

### Setup

In order to interact with this feature you must first retrieve a particular load balancer, like so:

```
$loadBalancer = $service->loadBalancer('{id}');
```

### Retrieve LB stats

You can retrieve detailed stats about your load balancer, including the following information:

- `connectTimeOut` – Connections closed by this load balancer because the 'connect_timeout' interval was exceeded.
- `connectError` – Number of transaction or protocol errors in this load balancer.
- `connectFailure` – Number of connection failures in this load balancer.
- `dataTimedOut` – Connections closed by this load balancer because the 'timeout' interval was exceeded.
- `keepAliveTimedOut` – Connections closed by this load balancer because the 'keepalive_timeout' interval was exceeded.
- `maxConn` – Maximum number of simultaneous TCP connections this load balancer has processed at any one time.

```
/** @var $stats OpenCloud\LoadBalancer\Resource\Stats **/
$stats = $loadBalancer->stats();
```

### Usage Reports

The load balancer usage reports provide a view of all transfer activity, average number of connections, and number of virtual IPs associated with the load balancing service. Current usage represents all usage recorded within the preceding 24 hours. Values for both incomingTransfer and outgoingTransfer are expressed in bytes transferred.

The optional startTime and endTime parameters can be used to filter all usage. If the startTime parameter is supplied but the endTime parameter is not, then all usage beginning with the startTime will be provided. Likewise, if the endTime parameter is supplied but the startTime parameter is not, then all usage will be returned up to the endTime specified.

```
# View billable LBs
$billable = $service->billableLoadBalancerList();

foreach ($billable as $loadBalancer) {
    /** @var $loadBalancer OpenCloud\LoadBalancer\Resource\LoadBalancer **/

    # View usage
    /** @var $usage OpenCloud\LoadBalancer\Resource\UsageRecord **/
    $usage = $loadBalancer->usage();
```

```
    echo $usage->averageNumConnections, PHP_EOL;
}
```

## Glossary

**allowed domain** Allowed domains are a restricted set of domain names that are allowed to add load balancer nodes.

**content caching** When content caching is enabled on a load balancer, recently-accessed files are stored on the load balancer for easy retrieval by web clients. Requests to the load balancer for these files are serviced by the load balancer itself, which reduces load off its back-end nodes and improves response times as well.

**health monitor** The load balancing service includes a health monitoring operation which periodically checks your back-end nodes to ensure they are responding correctly. If a node is not responding, it is removed from rotation until the health monitor determines that the node is functional. In addition to being performed periodically, the health check also is performed against every node that is added to ensure that the node is operating properly before allowing it to service traffic. Only one health monitor is allowed to be enabled on a load balancer at a time.

**load balancer** A load balancer is a device that distributes incoming network traffic amongst multiple back-end systems. These back-end systems are called the nodes of the load balancer.

**metadata** Metadata can be associated with each load balancer and each node for the client's personal use. It is defined using key-value pairs where the key and value consist of alphanumeric characters. A key is unique per load balancer.

**node** A node is a backend device that provides a service on specified IP and port. An example of a load balancer node might be a web server serving HTTP traffic on port 8080. A load balancer typically has multiple nodes attached to it so it can distribute incoming network traffic amongst them.

**session persistence** Session persistence is a feature of the load balancing service that forces multiple requests, of the same protocol, from clients to be directed to the same node. This is common with many web applications that do not inherently share application state between back-end servers.

**virtual IP** A virtual IP (VIP) makes a load balancer accessible by clients. The load balancing service supports either a public VIP address (`PUBLIC`), routable on the public Internet, or a ServiceNet VIP address (`SERVICENET`), routable only within the region in which the load balancer resides.

## Further Links

- API Developer Guide
- API release history

# Monitoring v1

**Note:** This service is only available for Rackspace users.

## Setup

The first step is to pass in your credentials and set up a client. For Rackspace users, you will need your username and API key:

```
use OpenCloud\Rackspace;

$client = new Rackspace(Rackspace::US_IDENTITY_ENDPOINT, array(
  'username' => '{username}',
  'apiKey'   => '{apiKey}',
));
```

### Monitoring service

Now to instantiate the Monitoring service:

```
$service = $client->monitoringService('{catalogName}', '{region}', '{urlType}');
```

- `{catalogName}` is the name of the service as it appears in the service catalog. OpenStack users *must* set this value. For Rackspace users, a default will be provided if you pass in `null`.
- `{region}` is the region the service will operate in. For Rackspace users, you can select one of the following from the *supported regions page*.
- `{urlType}` is the *type of URL* to use, depending on which endpoints your catalog provides. If omitted, it will default to the public network.

## Operations

### Entities

An entity is the target of what you are monitoring. For example, you can create an entity to monitor your website, a particular web service, or your Rackspace server. Note that an entity represents only one item in the monitoring system – if you wanted to monitor each server in a cluster, you would create an entity for each of the servers. You would not create a single entity to represent the entire cluster.

An entity can have multiple checks associated with it. This allows you to check multiple services on the same host by creating multiple checks on the same entity, instead of multiple entities each with a single check.

### Create Entity

```
$service->createEntity(array(
    'label' => 'Brand New Entity',
    'ip_addresses' => array(
        'default' => '127.0.0.4',
        'b'       => '127.0.0.5',
        'c'       => '127.0.0.6',
        'test'    => '127.0.0.7'
    ),
    'metadata' => array(
        'all'  => 'kinds',
        'of'   => 'stuff',
        'can'  => 'go',
        'here' => 'null is not a valid value'
```

```
    )
));
```

### Retrive an entity

```
$entity = $service->getEntity('{entityId}');
```

### Attributes

| Name | Description | Data type | Method |
|---|---|---|---|
| label | Defines a name for the entity. | String (1..255 chars) | `getLabel()` |
| agent_id | Agent to which this entity is bound to. | String matching the regex: `/^[-\.\w]{1,255}$/` | `getAgentId()` |
| ip_addresses | Hash of IP addresses that can be referenced by checks on this entity. | Array | `getIpAddresses()` |
| meta-data | Arbitrary key/value pairs that are passed during the alerting phase. | `OpenCloud\Common\Metadata` | `getMetadata()` |

### Update an entity

```
$entity->update(array(
    'label' => 'New label for my entity'
));
```

### Delete entity

```
$entity->delete();
```

### Checks

A check is one of the foundational building blocks of the monitoring system. The check determines the parts or pieces of the entity that you want to monitor, the monitoring frequency, how many monitoring zones are originating the check, and so on. When you create a new check in the monitoring system, you specify the following information:

- A name for the check
- The check's parent entity
- The type of check you're creating
- Details of the check
- The monitoring zones that will launch the check

The check, as created, will not trigger alert messages until you create an alarm to generate notifications, to enable the creation of a single alarm that acts upon multiple checks (e.g. alert if any of ten different servers stops responding) or multiple alarms off of a single check. (e.g. ensure both that a HTTPS server is responding and that it has a valid certificate).

### Create a check

There are various attributes available to you when creating a new monitoring check:

```
$params = array(
    'type'    => 'remote.http',
    'details' => array(
        'url'    => 'http://example.com',
        'method' => 'GET'
    ),
    'monitoring_zones_poll' => array('mzlon'),
    'period' => '100',
    'timeout' => '30',
    'target_alias' => 'default',
    'label'  => 'Website check 1'
);
```

For a full list of available attributes, consult the list below.

### Attributes

| Name | Description | Required? | Data type |
|---|---|---|---|
| type | The type of check. | Required | Valid check type. String (1..25 chars) |
| details | Details specific to the check type. | Optional | Array |
| disabled | Disables the check. | Optional | Boolean |
| label | A friendly label for a check. | Optional | String (1..255 chars) |
| metadata | Arbitrary key/value pairs. | Optional | Array |
| period | The period in seconds for a check. The value must be greater than the minimum period set on your account. | Optional | Integer (30..1800) |
| timeout | The timeout in seconds for a check. This has to be less than the period. | Optional | Integer (2..1800) |

### Optional attributes to be used with remote checks

| Name | Description | Required? | Data type |
|---|---|---|---|
| monitoring_zones_poll | List of monitoring zones to poll from. Note: This argument is only required for remote (non-agent) checks | Optional | Array |
| target_alias | A key in the entity's `ip_addresses` hash used to resolve this check to an IP address. This parameter is mutually exclusive with target_hostname. | Optional | String (1..64 chars) |
| target_hostname | The hostname this check should target. This parameter is mutually exclusive with `target_alias`. | Optional | Valid FQDN, IPv4 or IPv6 address. String (1..256 chars). |
| target_resolver | Determines how to resolve the check target. | Optional | `IPv4` or `IPv6` |

### Test parameters

Sometimes it can be useful to test out the parameters before sending them as a create call. To do this, pass in the `$params` like so:

```
$response = $entity->testNewCheckParams($params);

echo $response->timestamp; // When was it executed?
echo $response->available; // Was it available?
echo $response->status;    // Status code
```

### Send parameters

Once you are satisfied with your configuration parameters, you can complete the operation and send it to the API like so:

```
$entity->createCheck($params);
```

### Test existing Check

```
// Set arg to TRUE for debug information
$response = $check->test(true);

echo $response->debug_info;
```

### List Checks

```
$checks = $entity->getChecks();

foreach ($checks as $check) {
    echo $check->getId();
}
```

### Update Check

```
$check->update(array('period' => 500));
```

### Delete check

```
$check->delete();
```

### Check types

Each check within the Rackspace Cloud Monitoring has a designated check type. The check type instructs the monitoring system how to check the monitored resource. **Note:** Users cannot create, update or delete check types.

Check types for commonly encountered web protocols, such as HTTP (`remote.http`), IMAP (`remote.imap-banner`) , SMTP (`remote.stmp`), and DNS (`remote.dns`) are provided. Monitoring commonly encountered infrastructure servers like MySQL (`remote.mysql-banner`) and PostgreSQL (`remote.postgresql-banner`) are also available. Monitoring custom server uptime can be accomplished with the remote.tcp banner check to check for a protocol-defined banner at the beginning of a connection. Gathering metrics from server software to create alerts against can be accomplished using the remote.http check type and the 'extract' attribute to define the format.

In addition to the standard Cloud Monitoring check types, you can also use agent check types if the Monitoring Agent is installed on the server you are monitoring. For a list of available check types, see the official API documentation.

Checks generate metrics that alarms will alert based upon. The metrics generated often times depend on the check's parameters. For example, using the 'extract' attribute on the remote.http check, however the default metrics will always be present. To determine the exact metrics available, the Test Check API is provided.

### Find an existing check's type

If you want to see the type for an existing Check resource:

```
/** @var \OpenCloud\CloudMonitoring\Resource\CheckType */
$checkType = $check->getCheckType();
```

### List all possible check types

```
$checkTypes = $service->getCheckTypes();

foreach ($checkTypes as $checkType) {
   echo $checkType->getId();
}
```

### Retrieve details about a Type by its ID

Alternatively, you can retrieve a specific type based on its ID:

```
$checkTypeId = 'remote.dns';
$checkType = $service->getCheckType($checkTypeId);
```

### Attributes

Once you have access to a `OpenCloud\CloudMonitoring\Resource\CheckType` object, you can query these attributes:

| Name | Description | Data type | Method |
|---|---|---|---|
| type | The name of the supported check type. | String | `getType()` |
| fields | Check type fields. | Array | `getFields()` |
| supported_platforms | Platforms on which an agent check type is supported. This is advisory information only - the check may still work on other platforms, or report that check execution failed at runtime | Array | `getSupportedPlatforms()` |

### Alarms

Alarms bind alerting rules, entities, and notification plans into a logical unit. Alarms are responsible for determining a state (`OK`, `WARNING` or `CRITICAL`) based on the result of a Check, and executing a notification plan whenever that state changes. You create alerting rules by using the alarm DSL. For information about using the alarm language, refer to the reference documentation.

### Setup

In order to interact with this feature, you must first retrieve an entity by its ID:

```
$entity = $service->getEntity('{entityId}');
```

and then a particular check, about which you can configure alarms:

```
$check = $entity->getCheck('{checkId}');
```

For more information about these resource types, please consult the documentation about entities and checks.

### Retrieve alarm

```
$alarm = $check->getAlarm('{alarmId}');
```

Once you have access to a `OpenCloud\Monitoring\Resource\Alarm` object, these are the attributes you can access:

| Name | Description | Required? | Method |
|---|---|---|---|
| check_id | The ID of the check to alert on. | Required | `getCheckId()` |
| notification_plan_id | The ID of the notification plan to execute when the state changes. | Optional | `getNotificationPlanId()` |
| criteria | The alarm DSL for describing alerting conditions and their output states. | Optional | `getCriteria()` |
| disabled | Disable processing and alerts on this alarm | Optional | `isDisabled()` <bool> |
| label | A friendly label for an alarm. | Optional | `getLabel()` |
| metadata | Arbitrary key/value pairs. | Optional | `getMetadata()` |

### Create Alarm

```
$alarm = $check->getAlarm();
$alarm->create(array(
    'check_id' => 'chAAAA',
    'criteria' => 'if (metric["duration"] >= 2) { return new AlarmStatus(OK); }␣
↪return new AlarmStatus(CRITICAL);',
    'notification_plan_id' => 'npAAAAA'
));
```

### List Alarms

```
$alarms = $entity->getAlarms();

foreach ($alarms as $alarm) {
    echo $alarm->getId();
}
```

### Update Alarm

```
$alarm->update(array(
    'criteria' => 'if (metric["duration"] >= 5) { return new AlarmStatus(OK); }␣
↪return new AlarmStatus(CRITICAL);'
));
```

### Delete alarm

```
$alarm->delete();
```

### Agents

The Monitoring Agent resides on the host server being monitored. The agent allows you to gather on-host metrics based on agent checks and push them to Cloud Monitoring where you can analyze them, use them with the Cloud Monitoring infrastructure (such as alarms), and archive them.

For more information about this feature, including a brief overview of its core design principles and security layers, see the official API documentation.

### Retrieve details about an agent

```
$agent = $service->getAgent('{agentId}');
```

### List agents

```
$agents = $service->getAgents();

foreach ($agents as $agent) {
   echo $agent->getLastConnected();
}
```

### List connections

```
$connections = $agent->getConnections();
```

### Get connection

```
/** @var \OpenCloud\CloudMonitoring\Resource\AgentConnection */
$connection = $agent->getConnection('{connectionId}');
```

Once you have access to an agent's `OpenCloud\CloudMonitoring\Resource\AgentConnection` object, these are the attributes you can access:

| Name | Method |
|---|---|
| id | `getId()` |
| guid | `getGuid()` |
| agent_id | `getAgentId()` |
| endpoint | `getEndpoint()` |
| process_version | `getProcessVersion()` |
| bundle_version | `getBundleVersion()` |
| agent_ip | `getAgentIp()` |

### Agent tokens

Agent tokens are used to authenticate Monitoring agents to the Monitoring Service. Multiple agents can share a single token.

### Retrieve an agent token

```
$agentToken = $service->getAgentToken('{tokenId}');
```

### Create agent token

```
$newToken = $service->getAgentToken();
$newToken->create(array('label' => 'Foobar'));
```

### List agent tokens

```
$agentTokens = $service->getAgentTokens();

foreach ($agentTokens as $token) {
    echo $token->getLabel();
}
```

### Update agent token

```
$token->update(array(
    'label' => 'New label'
));
```

### Update agent token

```
$token->delete();
```

### Agent Host Information

An agent can gather host information, such as process lists, network configuration, and memory usage, on demand. You can use the host-information API requests to gather this information for use in dashboards or other utilities.

### Setup

```
$host = $service->getAgentHost();
```

### Get some metrics

```
$cpuInfo        = $host->info('cpus');
$diskInfo       = $host->info('disks');
$filesystemInfo = $host->info('filesystems');
$memoryInfo     = $host->info('memory');
$networkIntInfo = $host->info('network_interfaces');
$processesInfo  = $host->info('processes');
$systemInfo     = $host->info('system');
$userInfo       = $host->info('who');

// What CPU models do we have?
foreach ($cpuInfo as $cpuMetric) {
    echo $cpuMetric->model, PHP_EOL;
}

// How many disks do we have?
echo $diskInfo->count();

// What's the available space on our ext4 filesystem?
foreach ($filesystemInfo as $filesystemMetric) {
    if ($filesystemMetric->sys_type_name == 'ext4') {
        echo $filesystemMetric->avail;
    }
}
```

### Agent targets

Each agent check type gathers data for a related set of target devices on the server where the agent is installed. For example, `agent.network` gathers data for network devices. The actual list of target devices is specific to the configuration of the host server. By focusing on specific targets, you can efficiently narrow the metric data that the agent gathers.

### List agent targets

```
$targets = $service->getAgentTargets();

foreach ($targets as $target) {
    echo $target->getType();
}
```

### Changelogs

The monitoring service records changelogs for alarm statuses. Changelogs are accessible as a Time Series Collection. By default the API queries the last 7 days of changelog information.

### View Changelog

```
$changelog = $service->getChangelog();

foreach ($changelog as $item) {
   $entity = $item->getEntityId();
}
```

### Metrics

When Monitoring checks run, they generate metrics. These metrics are stored as full resolution data points in the Cloud Monitoring system. Full resolution data points are periodically rolled up (condensed) into coarser data points.

Depending on your needs, you can use the metrics API to fetch individual data points (fine-grained) or rolled up data points (coarse-grained) over a period of time.

### Data Granularity

Cloud Monitoring supports several granularities of data: full resolution data and rollups computed at 5, 20, 60, 240 and 1440 minute intervals.

When you fetch metrics data points, you specify several parameters to control the granularity of data returned:

- A time range for the points
- Either the number of points you want returned OR the resolution of the data you want returned

When you query by points, the API selects the resolution that will return you the number of points you requested. The API makes the assumption of a 30 second frequency, performs the calculation, and selects the appropriate resolution.

**Note:** Because the API performs calculations to determine the points returned for a particular resolution, the number of points returned may differ from the specific number of points you request.

Consider that you want to query data for a 48-hour time range between the timestamps `from=1354647221000` and `to=1358794421000` ( **specified in Unix time, based on the number of milliseconds that have elapsed since January 1, 1970** ). The following table shows the number of points that the API returns for a given resolution.

### Specifying resolution to retrieve data in 48 hour period

| You specify resolution... | API returns points... |
|---|---|
| FULL | 5760 |
| MIN5 | 576 |
| MIN20 | 144 |
| MIN60 | 48 |
| MIN240 | 12 |
| MIN1440 | 2 |

### Specifying number of points to retrieve data in 48 hour period

| You specify points in the range... | API calculates resolution |
|---|---|
| 3168-∞ | FULL |
| 360-3167 | MIN5 |
| 96-359 | MIN20 |
| 30-95 | MIN60 |
| 7-29 | MIN240 |
| 0-6 | MIN1440 |

### Data Point Expiration

Cloud Monitoring expires data points according to the following schedule:

| Resolution | Expiration |
|---|---|
| FULL | 2 days |
| MIN5 | 7 days |
| MIN20 | 15 days |
| MIN60 | 30 days |
| MIN240 | 60 days |
| MIN1440 | 365 days |

### Setup

In order to interact with this feature, you must first retrieve an entity by its ID:

```
$entity = $service->getEntity('{entityId}');
```

and then a particular check, about which you can configure alarms:

```
$check = $entity->getCheck('{checkId}');
```

For more information about these resource types, please consult the documentation about entities and checks.

### List all metrics

```
$metrics = $check->getMetrics();

foreach ($metrics as $metric) {
```

```
    echo $metric->getName();
}
```

### Fetch data points

```
$data = $check->fetchDataPoints('mzdfw.available', array(
    'resolution' => 'FULL',
    'from'       => 1369756378450,
    'to'         => 1369760279018
));
```

### Notifications

A notification is a destination to send an alarm; it can be a variety of different types, and will evolve over time.

For instance, with a webhook type notification, Cloud Monitoring posts JSON formatted data to a user-specified URL on an alert condition (Check goes from `OK` -> `CRITICAL` and so on).

### Get notification

```
$notification = $service->getNotification('{id}');
```

Once you have access to a `OpenCloud\Monitoring\Resource\Notification` object, these are the attributes available for use:

| Name | Description | Data type | Method |
|---|---|---|---|
| details | A hash of notification specific details based on the notification type. | Array | `getDetails()` |
| label | Friendly name for the notification. | String (1..255 chars) | `getLabel()` |
| type | The notification type to send. | String. Either `webhook`, `email`, or `pagerduty` | `getType()` |

### Creating notifications

The first thing to do when creating a new notification is configure the parameters which will define the behaviour of your resource:

```
$params = array(
    'label' => 'My webhook #1',
    'type'  => 'webhook',
    'details' => array(
        'url' => 'http://example.com'
    )
);
```

### Test parameters

Once this is done, it is often useful to test them out to check whether they will result in a successful creation:

```
// Test it
$response = $notification->testParams($params);

if ($response->status == 'Success') {
    echo $response->message;
}
```

### Send parameters

When you're happy with the parameters you've defined, you can complete the operation by sending them to the API like so:

```
$notification->create($params);
```

### Test existing notification

```
$response = $notification->testExisting(true);
echo $response->debug_info;
```

### List Notifications

```
$notifications = $service->getNotifications();

foreach ($notifications as $notification) {
    echo $notification->getId();
}
```

### Update a Notification

```
$notification->update(array(
    'label' => 'New notification label'
));
```

### Delete a Notification

```
$notification->delete();
```

### Notification types

Rackspace Cloud Monitoring currently supports the following notification types:

Industry-standard web hooks, where JSON is posted to a configurable URL. It has these attributes:

| Name | Description | Data type |
|---|---|---|
| address | Email address to send notifications to | Valid email |

Email alerts where the message is delivered to a specified address. It has these attributes:

| Name | Description | Data type |
|---|---|---|
| url | An HTTP or HTTPS URL to POST to | Valid URL |

### Setup

If you've already set up a main Notification object, and want to access functionality for this Notification's particular Notification Type, you can access its property:

```
$type = $notification->getNotificationType();
```

Alternatively, you can retrieve an independent resource using the ID:

```
$typeId = 'pagerduty';
$type = $service->getNotificationType($typeId);
```

### List all possible notification types

```
$types = $service->getNotificationTypes();

foreach ($types as $type) {
    echo sprintf('%s %s', $type->getName(), $type->getDescription());
}
```

### Notification plans

A notification plan contains a set of notification actions that Rackspace Cloud Monitoring executes when triggered by an alarm. Rackspace Cloud Monitoring currently supports webhook and email notifications.

Each notification state can contain multiple notification actions. For example, you can create a notification plan that hits a webhook/email to notify your operations team if a warning occurs. However, if the warning escalates to an Error, the notification plan could be configured to hit a different webhook/email that triggers both email and SMS messages to the operations team. The notification plan supports the following states:

- Critical
- Warning
- OK

A notification plan, `npTechnicalContactsEmail`, is provided by default which will email all of the technical contacts on file for an account whenever there is a state change.

### Get a notification plan

```
$plan = $service->getNotificationPlan('{planId}');
```

Once you have access to a `OpenCloud\\Monitoring\\Resource\\NotificationPlan` object, you can access these resources:

| Name | Description | Required? | Data type | Method |
|---|---|---|---|---|
| label | Friendly name for the notification plan. | Required | String (1..255 chars) | `getLabel()` |
| critical_state | The notification list to send to when the state is `CRITICAL`. | Optional | Array | `getCriticalState()` |
| ok_state | The notification list to send to when the state is `OK`. | Optional | Array | `getOkState()` |
| warning_state | The notification list to send to when the state is `WARNING`. | Optional | Array | `getWarningState()` |

### Create Notification Plan

```
$plan->create(array(
    'label'          => 'New Notification Plan',
    'critical_state' => array('ntAAAA'),
    'ok_state'       => array('ntBBBB'),
    'warning_state'  => array('ntCCCC')
));
```

### Update notification plan

```
$plan->update(array(
    'label' => 'New label for my plan'
));
```

### Delete notification plan

```
$plan->delete();
```

### Alarm Notification History

The monitoring service keeps a record of notifications sent for each alarm. This history is further subdivided by the check on which the notification occurred. Every attempt to send a notification is recorded, making this history a valuable tool in diagnosing issues with unreceived notifications, in addition to offering a means of viewing the history of an alarm's statuses.

Alarm notification history is accessible as a Time Series Collection. By default alarm notification history is stored for 30 days and the API queries the last 7 days of information.

### Setup

In order to interact with this feature, you must first retrieve an entity by its ID:

```
$entity = $service->getEntity('{entityId}');
```

and then a particular check, about which you can configure alarms:

```
$check = $entity->getCheck('{checkId}');
```

and finally, retrieve the alarm:

```
$alarm = $check->getAlarm('{alarmId}');
```

For more information about these resource types, please consult the documentation about entities and checks.

### Discover which Checks have a Notification History

This operation list checks for which alarm notification history is available:

```
$checks = $alarm->getRecordedChecks();
```

### List Alarm Notification History for a particular Check

```
$checkHistory = $alarm->getNotificationHistoryForCheck('chAAAA');
```

### Get a particular Notification History item

```
$checkId  = 'chAAAA';
$itemUuid = '646ac7b0-0b34-11e1-a0a1-0ff89fa2fa26';

$singleItem = $history->getNotificationHistoryItem($checkId, $itemUuid);
```

### Views

Views contain a combination of data that usually includes multiple, different objects. The primary purpose of a view is to save API calls and make data retrieval more efficient. Instead of doing multiple API calls and then combining the result yourself, you can perform a single API call against the view endpoint.

### List all Views

```
$views = $service->getViews();

foreach ($views as $view) {
    $entity = $view->getEntity();
    echo $view->getTimestamp();
}
```

### Zones

A monitoring zone is a location that Rackspace Cloud Monitoring collects data from. Examples of monitoring zones are “US West”, “DFW1” or “ORD1”. It is an abstraction for a general location from which data is collected.

An “endpoint,” also known as a “collector,” collects data from the monitoring zone. The endpoint is mapped directly to an individual machine or a virtual machine. A monitoring zone contains many endpoints, all of which will be within

the IP address range listed in the response. The opposite is not true, however, as there may be unallocated IP addresses or unrelated machines within that IP address range.

A check references a list of monitoring zones it should be run from.

### Get details about a zone

```
$zone = $monitoringService->getMonitoringZone('{zoneId}');
```

| Name | Description | Data type | Method |
|---|---|---|---|
| country_code | Country Code | String longer than 2 characters | `getCountryCode()` |
| label | Label | String | `getLabel()` |
| source_ips | Source IP list | Array | `getSourceIps()` |

### List all zones

```
$zones = $service->getMonitoringZones();
```

### Perform a traceroute

```
$traceroute = $zone->traceroute(array(
    'target' => 'http://test.com',
    'target_resolver' => 'IPv4'
));

// How many hops?
echo count($traceroute);

// What was the first hop's IP?
echo $traceroute[0]->ip;
```

## Glossary

**agent** A monitoring daemon that resides on the server being monitored. The agent gathers metrics based on agent checks and pushes them to Cloud Monitoring. The agent provides insight into your servers with checks for information such as load average and network usage. The agent acts as a single small service that runs scheduled checks and pushes metrics to the rest of Cloud Monitoring so the metrics can be analyzed, trigger alerts, and be archived. These metrics are gathered via checks using agent check types, and can be used with the other Cloud Monitoring primitives such as alarms.

**agent token** An authentication token used to identify the agent when it communicates with Cloud Monitoring.

**alarm** An alarm contains a set of rules that determine when the monitoring system sends a notification. You can create multiple alarms for the different checks types associated with an entity. For example, if your entity is a web server that hosts your company's website, you can create one alarm to monitor the server itself, and another alarm to monitor the website.

**check** Checks explicitly specify how you want to monitor an entity. Once you've created an entity, you can configure one or more checks for it. A check is the foundational building block of the monitoring system, and is always associated with an entity. The check specifies the parts or pieces of the entity that you want to monitor, the

monitoring frequency, how many monitoring zones are launching the check, and so on. It contains the specific details of how you are monitoring the entity.

**entity** The object or resource that you want to monitor. It can be any object or device that you want to monitor. It's commonly a web server, but it might also be a website, a web page or a web service.

**monitoring zone** A monitoring zone is the "launch point" of a check. When you create a check, you specify which monitoring zone(s) you want to launch the check from. This concept of a monitoring zone is similar to that of a datacenter, however in the monitoring system, you can think of it more as a geographical region.

**notification** A notification is an informational message sent to one or more addresses by the monitoring system when an alarm is triggered. You can set up notifications to alert a single individual or an entire team. Rackspace Cloud Monitoring currently supports webhooks and email for sending notifications.

**notification plan** A notification plan contains a set of notification rules to execute when an alarm is triggered. A notification plan can contain multiple notifications for each of the following states:

### Further links

- Getting Started Guide for the API
- API Developer Guide
- API release history

## Networking v2

### Setup

#### Rackspace setup

The first step is to pass in your credentials and set up a client. For Rackspace users, you will need your username and API key:

```
use OpenCloud\Rackspace;

$client = new Rackspace(Rackspace::US_IDENTITY_ENDPOINT, array(
  'username' => '{username}',
  'apiKey'   => '{apiKey}',
));
```

#### OpenStack setup

If you're an OpenStack user, you will also need to prove a few other configuration parameters:

```
$client = new OpenCloud\OpenStack('{keystoneUrl}', array(
  'username' => '{username}',
  'password' => '{apiKey}',
  'tenantId' => '{tenantId}',
));
```

### Networking service

Now to instantiate the Networking service:

```
$service = $client->networkingService('{catalogName}', '{region}', '{urlType}');
```

- `{catalogName}` is the name of the service as it appears in the service catalog. OpenStack users *must* set this value. For Rackspace users, a default will be provided if you pass in `null`.
- `{region}` is the region the service will operate in. For Rackspace users, you can select one of the following from the *supported regions page*.
- `{urlType}` is the *type of URL* to use, depending on which endpoints your catalog provides. If omitted, it will default to the public network.

## Operations

### Networks

### Create a network

This operation takes one parameter, an associative array, with the following keys:

| Name | Description | Data type | Re-quired? | Default value | Example value |
|---|---|---|---|---|---|
| `name` | A human-readable name for the network. This name might not be unique. | String | No | `null` | `My private backend network` |
| `adminStateUp` | The administrative state of network. If `false` (down), the network does not forward packets. | Boolean | No | `true` | `true` |
| `shared` | Specifies whether the network resource can be accessed by any tenant. | Boolean | No | `false` | `false` |
| `tenantId` | Owner of network. Only admin users can specify a tenant ID other than their own. | String | No | Same as tenant creating the network | `123456` |

You can create a network as shown in the following example:

```
/** @var $network OpenCloud\Networking\Resource\Network **/
$network = $networkingService->createNetwork(array(
    'name' => 'My private backend network'
));
```

Get the executable PHP script for this example

### Create multiple networks

This operation takes one parameter, an indexed array. Each element of this array must be an associative array with the keys shown in *the preceding table*.

You can create multiple networks as shown in the following example:

```
$networks = $networkingService->createNetworks(array(
    array(
        'name' => 'My private backend network #1'
    ),
    array(
        'name' => 'My private backend network #2'
    )
));

foreach ($networks as $network) {
    /** @var $network OpenCloud\Networking\Resource\Network **/
}
```

Get the executable PHP script for this example

### List networks

You can list all the networks to which you have access as shown in the following example:

```
$networks = $networkingService->listNetworks();

foreach ($networks as $network) {
    /** @var $network OpenCloud\Networking\Resource\Network **/
}
```

Get the executable PHP script for this example

### Get a network

You can retrieve a specific network by using that network's ID, as shown in the following example:

```
/** @var $network OpenCloud\Networking\Resource\Network **/
$network = $networkingService->getNetwork('{networkId}');
```

Get the executable PHP script for this example

### Update a network

This operation takes one parameter, an associative array, with the following keys:

| Name | Description | Data type | Required? | Default value | Example value |
|---|---|---|---|---|---|
| `name` | A human-readable name for the network. This name might not be unique. | String | No | `null` | `My updated private backend network` |
| `adminStateUp` | The administrative state of network. If `false` (down), the network does not forward packets. | Boolean | No | `true` | `true` |
| `shared` | Specifies whether the network resource can be accessed by any tenant. | Boolean | No | `false` | `false` |

You can update a network as shown in the following example:

```
$network->update(array(
    'name' => 'My updated private backend network'
));
```

Get the executable PHP script for this example

### Delete a network

You can delete a network as shown in the following example:

```
$network->delete();
```

Get the executable PHP script for this example

### Subnets

### Create a subnet

This operation takes one parameter, an associative array, with the following keys:

| Name | Description | Data type | Required? | Default value | Example value |
|---|---|---|---|---|---|
| networkId | Network this subnet is associated with | String | Yes | • | eb60583c-57ea-41b9-8d5c- |
| ipVersion | IP version | Integer (4 or 6) | Yes | • | 4 |
| cidr | CIDR representing the IP address range for this subnet | String (CIDR) | Yes | • | 192.168.199.0/25 |
| name | A human-readable name for the subnet. This name might not be unique. | String | No | null | My subnet |
| gatewayIp | IP address of the default gateway used by devices on this subnet | String (IP address) | No | First IP address in CIDR | 192.168.199.128 |
| dnsNameservers | DNS nameservers used by hosts in this subnet | Indexed array of strings | No | Empty array | array('4.4.4.4', '8.8.8.8') |
| allocationPools | Subranges of the CIDR available for dynamic allocation to ports | Indexed array of associative arrays | No | Every IP address in CIDR, excluding gateway IP address if configured | array(array('start' => '192.168.199.2', 'end' => '192.168.199.127')) |
| hostRoutes | Routes that should be used by devices with IP addresses from this subnet (not including the local subnet route) | Indexed array of associative arrays | No | Empty array | array(array('destination => '1.1.1.0/24', 'nexthop' => '192.168.19.20')) |
| enableDhcp | Specifies whether DHCP is enabled for this subnet | Boolean | No | true | false |
| tenantId | Owner of the subnet. Only admin users can specify a tenant ID other than their own. | String | No | Same as tenant creating the subnet | 123456 |

You can create a subnet as shown in the following example:

```
/** @var $subnet OpenCloud\Networking\Resource\Subnet **/
$subnet = $networkingService->createSubnet(array(
```

```
    'name' => 'My subnet',
    'networkId' => 'eb60583c-57ea-41b9-8d5c-8fab2d22224c',
    'ipVersion' => 4,
    'cidr' => '192.168.199.0/25'
));
```

Get the executable PHP script for this example

### Create multiple subnets

This operation takes one parameter, an indexed array. Each element of this array must be an associative array with the keys shown in *the preceding table*.

You can create multiple subnets as shown in the following example:

```
$subnets = $networkingService->createSubnets(array(
    array(
        'name' => 'My subnet #1'
    ),
    array(
        'name' => 'My subnet #2'
    )
));

foreach ($subnets as $subnet) {
    /** @var $subnet OpenCloud\Networking\Resource\Subnet **/
}
```

Get the executable PHP script for this example

### List subnets

You can list all the subnets to which you have access as shown in the following example:

```
$subnets = $networkingService->listSubnets();
foreach ($subnets as $subnet) {
    /** @var $subnet OpenCloud\Networking\Resource\Subnet **/
}
```

Get the executable PHP script for this example

### Get a subnet

You can retrieve a specific subnet by using that subnet's ID, as shown in the following example:

```
/** @var $subnet OpenCloud\Networking\Resource\Subnet **/
$subnet = $networkingService->getSubnet('{subnetId}');
```

Get the executable PHP script for this example

### Update a subnet

This operation takes one parameter, an associative array, with the following keys:

| Name | Description | Data type | Required? | Default value | Example value |
|---|---|---|---|---|---|
| `name` | A human-readable name for the subnet. This name might not be unique. | String | No | `null` | `My updated subnet` |
| `gatewayIp` | IP address of the default gateway used by devices on this subnet | String (IP address) | No | First IP address in CIDR | `192.168.62.155` |
| `dnsNameservers` | DNS nameservers used by hosts in this subnet | Indexed array of strings | No | Empty array | `array('4.4.4.4', '8.8.8.8')` |
| `hostRoutes` | Routes that should be used by devices with IP adresses from this subnet (not including the local subnet route) | Indexed array of associative arrays | No | Empty array | `array(array('destination' => '1.1.1.0/24', 'nexthop' => '192.168.17.19'))` |
| `enableDhcp` | Specifies whether DHCP is enabled for this subnet | Boolean | No | `true` | `false` |

You can update a subnet as shown in the following example:

```
$subnet->update(array(
    'name' => 'My updated subnet',
    'hostRoutes' => array(
        array(
            'destination' => '1.1.1.0/24',
            'nexthop'     => '192.168.17.19'
        )
    ),
    'gatewayIp' => '192.168.62.155'
));
```

Get the executable PHP script for this example

### Delete a subnet

You can delete a subnet as shown in the following example:

```
$subnet->delete();
```

Get the executable PHP script for this example

### Ports

### Create a port

This operation takes one parameter, an associative array, with the following keys:

| Name | Description | Data type | Required? | Default value | Example value |
|---|---|---|---|---|---|
| networkId | Network this port is associated with | String | Yes | • | eb60583c-57ea-41b9-8d5c- |
| name | A human-readable name for the port. This name might not be unique. | String | No | null | My port |
| adminStateUp | The administrative state of port. If false (down), the port does not forward packets. | Boolean | No | true | true |
| macAddress | MAC address to use on this port | String (MAC address in 6-octet form separated by colons) | No | Generated | 0F:5A:6F:70:E9:5C |
| fixedIps | IP addresses for this port | Indexed array of associative arrays | No | Automatically allocated from the pool | array(array('subnetId' => '75906d20-6625-11e4-9803 'ipAddress' => '192. 168.199. 17')) |
| deviceId | Identifies the device (for example, virtual server) using this port | String | No | null | 5e3898d7-11be-483e-9732- |
| deviceOwner | Identifies the entity (for example, DHCP agent) using this port | String | No | null | network:router_interface |
| securityGroups | Specifies the IDs of any security groups associated with this port | Indexed array of strings | No | Empty array | array('f0ac4394-7e4a-440 |
| tenantId | Owner of the port. Only admin users can specify a tenant ID other than their own. | String | No | Same as the tenant creating the port | 123456 |

You can create a port as shown in the following example:

```
/** @var $port OpenCloud\Networking\Resource\Port **/
$port = $networkingService->createPort(array(
    'name' => 'My port',
    'networkId' => 'eb60583c-57ea-41b9-8d5c-8fab2d22224c'
));
```

Get the executable PHP script for this example

### Create multiple ports

This operation takes one parameter, an indexed array. Each element of this array must be an associative array with the keys shown in *the preceding table*.

You can create multiple ports as shown in the following example:

```
$ports = $networkingService->createPorts(array(
    array(
        'name' => 'My port #1',
        'networkId' => 'eb60583c-57ea-41b9-8d5c-8fab2d22224c'
    ),
    array(
        'name' => 'My port #2',
        'networkId' => 'eb60583c-57ea-41b9-8d5c-8fab2d22224c'
    )
));

foreach ($ports as $port) {
    /** @var $port OpenCloud\Networking\Resource\Port **/
}
```

Get the executable PHP script for this example

### List ports

You can list all the ports to which you have access as shown in the following example:

```
$ports = $networkingService->listPorts();

foreach ($ports as $port) {
    /** @var $port OpenCloud\Networking\Resource\Port **/
}
```

Get the executable PHP script for this example

The port list query may be filtered by numerous optional parameters as per the API documentation

```
$ports = $networkingService->listPorts([
    'status' => 'ACTIVE',
    'device_id' => '9ae135f4-b6e0-4dad-9e91-3c223e385824'
]);

foreach ($ports as $port) {
    /** @var $port OpenCloud\Networking\Resource\Port **/
}
```

Get the executable PHP script for this example

### Get a port

You can retrieve a specific port by using that port's ID, as shown in the following example:

```
/** @var $port OpenCloud\Networking\Resource\Port **/
$port = $networkingService->getPort('{portId}');
```

Get the executable PHP script for this example

### Update a port

This operation takes one parameter, an associative array, with the following keys:

| Name | Description | Data type | Required? | Default value | Example value |
|---|---|---|---|---|---|
| `name` | A human-readable name for the port. This name might not be unique. | String | No | `null` | `My port` |
| `adminStateUp` | The administrative state of port. If `false` (down), the port does not forward packets. | Boolean | No | `true` | `true` |
| `fixedIps` | IP addresses for this port | Indexed array of associative arrays | No | Automatically allocated from the pool | `array(array('subnetId' => '75906d20-6625-11e4-9803-0800200c9a66', 'ipAddress' => '192.168.199.59'))` |
| `deviceId` | Identifies the device (for example, virtual server) using this port | String | No | `null` | `5e3898d7-11be-483e-9732-b2f5eccd2b2e` |
| `deviceOwner` | Identifies the entity (for example, DHCP agent) using this port | String | No | `null` | `network:router_interface` |
| `securityGroups` | Specifies the IDs of any security groups associated with this port | Indexed array of strings | No | Empty array | `array('f0ac4394-7e4a-4409-9701-ba8be283dbc` |

You can update a port as shown in the following example:

```
$port->update(array(
    'fixedIps' => array(
        array(
            'subnetId'  => '75906d20-6625-11e4-9803-0800200c9a66',
            'ipAddress' => '192.168.199.59'
        )
    )
));
```

Get the executable PHP script for this example

### Delete a port

You can delete a port as shown in the following example:

```
$port->delete();
```

Get the executable PHP script for this example

### Security Groups

#### Create a security group

This operation takes one parameter, an associative array, with the following keys:

| Name | Description | Data type | Required? | Default value | Example value |
|---|---|---|---|---|---|
| `name` | A human-readable name for the security group. This name might not be unique. | String | Yes | • | `new-webservers` |
| `description` | Description of the security group. | String | No | `null` | `security group for webservers` |

You can create a security group as shown in the following example:

```
/** @var $securityGroup OpenCloud\Networking\Resource\SecurityGroup **/
$securityGroup = $networkingService->createSecurityGroup(array(
    'name' => 'new-webservers',
    'description' => 'security group for webservers'
));
```

Get the executable PHP script for this example

#### List security groups

You can list all the security groups to which you have access as shown in the following example:

```
$securityGroups = $networkingService->listSecurityGroups();
foreach ($securityGroups as $securityGroup) {
    /** @var $securityGroup OpenCloud\Networking\Resource\SecurityGroup **/
}
```

Get the executable PHP script for this example

#### Get a security group

You can retrieve a specific security group by using that security group's ID, as shown in the following example:

```
/** @var $securityGroup OpenCloud\Networking\Resource\SecurityGroup **/
$securityGroup = $networkingService->getSecurityGroup('{secGroupId}');
```

Get the executable PHP script for this example

### Delete a security group

You can delete a security group as shown in the following example:

```
$securityGroup->delete();
```

Get the executable PHP script for this example

### Security Group Rules

### Create a security group rule

This operation takes one parameter, an associative array, with the following keys:

| Name | Description | Data type | Required? | Default value | Example value |
|---|---|---|---|---|---|
| securityGroupId | The security group ID to associate with this security group rule. | String | Yes | • | 2076db17-a522-4506-91de- |
| direction | The direction in which the security group rule is applied. For a compute instance, an ingress security group rule is applied to incoming (ingress) traffic for that instance. An egress rule is applied to traffic leaving the instance. | String (ingress or egress) | Yes | • | ingress |
| ethertype | Must be IPv4 or IPv6, and addresses represented in CIDR must match the ingress or egress rules. | String (IPv4 or IPv6) | No | IPv4 | IPv6 |
| portRangeMin | The minimum port number in the range that is matched by the security group rule. If the protocol is TCP or UDP, this value must be less than or equal to the value of the portRangeMax attribute. If the protocol is ICMP, this value must be an ICMP type. | Integer | No | null | 80 |
| portRangeMax | The maximum port number in the range that is matched by the security group rule. The port_range_min attribute constrains the attribute. If the protocol is ICMP, this value must be | Integer | No | null | 80 |

You can create a security group rule as shown in the following example:

```
/** @var $securityGroupRule OpenCloud\Networking\Resource\SecurityGroupRule **/
$securityGroupRule = $networkingService->createSecurityGroupRule(array(
    'securityGroupId' => '2076db17-a522-4506-91de-c6dd8e837028',
    'direction'       => 'egress',
    'ethertype'       => 'IPv4',
    'portRangeMin'    => 80,
    'portRangeMax'    => 80,
    'protocol'        => 'tcp',
    'remoteGroupId'   => '85cc3048-abc3-43cc-89b3-377341426ac5'
));
```

Get the executable PHP script for this example

### List security group rules

You can list all the security group rules to which you have access as shown in the following example:

```
$securityGroupRules = $networkingService->listSecurityGroupRules();
foreach ($securityGroupRules as $securityGroupRule) {
    /** @var $securityGroupRule OpenCloud\Networking\Resource\SecurityGroupRule **/
}
```

Get the executable PHP script for this example

## Glossary

**network** A network is an isolated virtual layer-2 broadcast domain that is typically reserved for the tenant who created it unless you configure the network to be shared. The network is the main entity in the Networking service. Ports and subnets are always associated with a network.

**subnet** A subnet represents an IP address block that can be used to assign IP addresses to virtual instances (such as servers created using the Compute service). Each subnet must have a CIDR and must be associated with a network.

**port** A port represents a virtual switch port on a logical network switch. Virtual instances (such as servers created using the Compute service) attach their interfaces into ports. The port also defines the MAC address and the IP address(es) to be assigned to the interfaces plugged into them. When IP addresses are associated to a port, this also implies the port is associated with a subet, as the IP address is taken from the allocation pool for a specific subnet.

**security group** A security group is a named container for security group rules.

**security group rule** A security group rule provides users the ability to specify the types of traffic that are allowed to pass through to and from ports on a virtual server instance.

## Further links

- Getting Started Guide for the API
- API Developer Guide
- API release history

# Object Store v1

## Setup

### Rackspace setup

The first step is to pass in your credentials and set up a client. For Rackspace users, you will need your username and API key:

```
use OpenCloud\Rackspace;

$client = new Rackspace(Rackspace::US_IDENTITY_ENDPOINT, array(
  'username' => '{username}',
  'apiKey'   => '{apiKey}',
));
```

### OpenStack setup

If you're an OpenStack user, you will also need to prove a few other configuration parameters:

```
$client = new OpenCloud\OpenStack('{keystoneUrl}', array(
  'username' => '{username}',
  'password' => '{apiKey}',
  'tenantId' => '{tenantId}',
));
```

### Object Store service

Now to instantiate the Object Store service:

```
$service = $client->objectStoreService('{catalogName}', '{region}', '{urlType}');
```

- `{catalogName}` is the name of the service as it appears in the service catalog. OpenStack users *must* set this value. For Rackspace users, a default will be provided if you pass in `null`.
- `{region}` is the region the service will operate in. For Rackspace users, you can select one of the following from the *supported regions page*.
- `{urlType}` is the *type of URL* to use, depending on which endpoints your catalog provides. If omitted, it will default to the public network.

## Operations

### Account Details

To see how many containers you have in your account (X-Account-Container-Count), how many objects are in your account (X-Account-Object-Count), and how many total bytes your account uses (X-Account-Bytes-Used):

### Setup

```
$account = $service->getAccount();
```

### View all details

```
$details = $account->getDetails();
```

### Retrieve total container count

```
$account->getContainerCount();
```

Get the executable PHP script for this example

### Retrieve total object count

```
$account->getObjectCount();
```

Get the executable PHP script for this example

### Retrieve total bytes used

```
$account->getBytesUsed();
```

Get the executable PHP script for this example

### Containers

### Create container

To create a new container, you just need to define its name:

```
$container = $service->createContainer('my_amazing_container');
```

If the response returned is `FALSE`, there was an API error - most likely due to the fact you have a naming collision.

Container names must be valid strings between 0 and 256 characters. Forward slashes are not currently permitted.

---

**Note:** When working with names that contain non-standard alphanumerical characters (such as spaces or non-English characters), you must ensure they are encoded with urlencode before passing them in

---

Get the executable PHP script for this example

### List containers

```
$containers = $service->listContainers();

foreach ($containers as $container) {
    /** @param $container OpenCloud\ObjectStore\Resource\Container */
    printf("Container name: %s\n", $container->name);
    printf("Number of objects within container: %d\n", $container->getObjectCount());
}
```

Container names are sorted based on a binary comparison, a single built-in collating sequence that compares string data using SQLite's memcmp() function, regardless of text encoding.

The list is limited to 10,000 containers at a time. To work with larger collections, please read the next section.

Get the executable PHP script for this example

### Filtering large collections

When you need more control over collections of containers, you can filter the results and return back a subset of the total collection by using the `marker` and `end_marker` parameters. The former parameter (`marker`) tells the API where to begin the list, and the latter (`end_marker`) tells it where to end the list. You may use either of them independently or together.

You may also use the `limit` parameter to fix the number of containers returned.

To list a set of containers between two fixed points:

```
$someContainers = $service->listContainers(array(
    'marker'     => 'container_55',
    'end_marker' => 'container_2001'
));
```

Or to return a limited set:

```
$someContainers = $service->listContainers(array('limit' => 560));
```

### Get container

To retrieve a certain container:

```
/** @param $container OpenCloud\ObjectStore\Resource\Container */
$container = $service->getContainer('{containerName}');
```

Get the executable PHP script for this example

### Retrieve a container's name

```
$name = $container->name;
```

### Retrieve a container's object count

```
$count = $container->getObjectCount();
```

Get the executable PHP script for this example

### Retrieve a container's total bytes used

```
$bytes = $container->getBytesUsed();
```

Get the executable PHP script for this example

### Delete container

Deleting an empty container is easy:

```
$container->delete();
```

Please bear mind that you must delete all objects inside a container before deleting it. This is done for you if you set the `$deleteObjects` parameter to `TRUE` like so:

```
$container->delete(true);
```

You can also *delete all objects* first, and then call `delete`.

Get the executable PHP script for this example

### Deleting all objects inside a container

```
$container->deleteAllObjects();
```

Get the executable PHP script for this example

### Create or update container metadata

```
$container->saveMetadata(array(
    'Author' => 'Virginia Woolf',
    'Published' => '1931'
));
```

Please bear in mind that this action will set metadata to this array - overriding existing values and wiping those left out. To *append* values to the current metadata:

```
$metadata = $container->appendToMetadata(array(
    'Publisher' => 'Hogarth'
));
```

Get the executable PHP script for this example

### Container quotas

The `container_quotas` middleware implements simple quotas that can be imposed on Cloud Files containers by a user. Setting container quotas 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.

To set quotas for a container:

```
use OpenCloud\Common\Constants\Size;

$container->setCountQuota(1000);
$container->setBytesQuota(2.5 * Size::GB);
```

And to retrieve them:

```
echo $container->getCountQuota();
echo $container->getBytesQuota();
```

Get the executable PHP scripts for this example:

- Set bytes quota
- Set count quota

### Access log delivery

To view your object access, turn on Access Log Delivery. You can use access logs to analyze the number of people who access your objects, where they come from, how many requests for each object you receive, and time-based usage patterns (such as monthly or seasonal usage).

```
$container->enableLogging();
$container->disableLogging();
```

### Syncing containers

You can synchronize local directories with your CloudFiles/Swift containers very easily. When you do this, the container will mirror exactly the nested file structure within your local directory:

```
$container->uploadDirectory('/home/user/my-blog');
```

There are four scenarios you should be aware of:

| Local | Remote | Comparison | Action |
|---|---|---|---|
| File exists | File exists | Identical checksum | No action |
| File exists | File exists | Different checksum | Local file overwrites remote |
| File exists | File does not exist | • | Local file created in Swift |
| Files does not exist | File exists | • | Remote file deleted |

### Objects

### Setup

In order to interact with this feature, you must first retrieve a particular container using its unique name:

```
$container = $service->getContainer('{containerName}');
```

### Create an object

There are three ways to upload a new file, each of which has different business needs.

---

**Note:** Unlike previous versions, you do not need to manually specify your object's content type. The API will do this for you.

---

---

**Note:** When working with names that contain non-standard alphanumerical characters (such as spaces or non-English characters), you must ensure they are encoded with urlencode before passing them in.

---

### Upload a single file (under 5GB)

The simplest way to upload a local object, without additional metadata, is by its path:

```
$container->uploadObject('example.txt', fopen('/path/to/file.txt', 'r+'));
```

The resource handle will be automatically closed by Guzzle in its destructor, so there is no need to execute `fclose`.

Get the executable PHP script for this example

### Upload a single file (under 5GB) with metadata

Although the previous section handles most use cases, there are times when you want greater control over what is being uploaded. For example, you might want to control the object's metadata, or supply additional HTTP headers to coerce browsers to handle the download a certain way. To add metadata to a new object:

```
use OpenCloud\ObjectStore\Resource\DataObject;

// specify optional metadata
$metadata = array(
    'Author' => 'Camera Obscura',
    'Origin' => 'Glasgow',
);

// specify optional HTTP headers
$httpHeaders = array(
    'Content-Type' => 'application/json',
);

// merge the two
$allHeaders = array_merge(DataObject::stockHeaders($metadata), $httpHeaders);
```

```
// upload as usual
$container->uploadObject('example.txt', fopen('/path/to/file.txt', 'r+'),
↪$allHeaders);
```

As you will notice, the first argument to `uploadObject` is the remote object name, i.e. the name it will be uploaded as. The second argument is either a file handle resource, or a string representation of object content (a temporary resource will be created in memory), and the third is an array of additional headers.

Get the executable PHP script for this example

### Batch upload multiple files (each under 5GB)

```
$files = array(
    array(
        'name' => 'apache.log',
        'path' => '/etc/httpd/logs/error_log'
    ),
    array(
        'name' => 'mysql.log',
        'body' => fopen('/tmp/mysql.log', 'r+')
    ),
    array(
        'name' => 'to_do_list.txt',
        'body' => 'PHONE HOME'
    )
);

$container->uploadObjects($files);
```

As you can see, the `name` key is required for every file. You must also specify *either* a path key (to an existing file), or a `body`. The `body` can either be a PHP resource or a string representation of the content you want to upload.

Get the executable PHP script for this example

### Upload large files (over 5GB)

For files over 5GB, you will need to use the `OpenCloud\ObjectStore\Upload\TransferBuilder` factory to build and execute your transfer. For your convenience, the Container resource object contains a simple method to do this heavy lifting for you:

```
$transfer = $container->setupObjectTransfer(array(
    'name'        => 'video.mov',
    'path'        => '/home/user/video.mov',
    'metadata'    => array('Author' => 'Jamie'),
    'concurrency' => 4,
    'partSize'    => 1.5 * Size::GB
));

$transfer->upload();
```

You can specify how many concurrent cURL connections are used to upload parts of your file. The file is fragmented into chunks, each of which is uploaded individually as a separate file (the filename of each part will indicate that it's a segment rather than the full file). After all parts are uploaded, a manifestfile is uploaded. When the end-user accesses

the 5GB by its true filename, it actually references the manifest file which concatenates each segment into a streaming download.

In Swift terminology, the name for this process is *Dynamic Large Object (DLO)*. To find out more details, please consult the official documentation.

Get the executable PHP script for this example

### List objects in a container

To return a list of objects:

```
$files = $container->objectList();

foreach ($files as $file) {
    /** @var $file OpenCloud\ObjectStore\Resource\DataObject */
}
```

By default, 10,000 objects are returned as a maximum. To get around this, you can construct a query which refines your result set. For a full specification of query parameters relating to collection filtering, see the official docs.

```
$container->objectList(array('prefix' => 'logFile_'));
```

Get the executable PHP script for this example

### List over 10,000 objects

To retrieve more than 10,000 objects (the default limit), you'll need to use the built-in paging which uses a 'marker' parameter to fetch the next page of data.

```
$containerObjects = array();
$marker = '';

while ($marker !== null) {
    $params = array(
        'marker' => $marker,
    );

    $objects = $container->objectList($params);
    $total = $objects->count();
    $count = 0;

    if ($total == 0) {
        break;
    }

    foreach ($objects as $object) {
        /** @var $object OpenCloud\ObjectStore\Resource\DataObject **/
        $containerObjects[] = $object->getName();
        $count++;

        $marker = ($count == $total) ? $object->getName() : null;
    }
}
```

Get the executable PHP script for this example

### Get object

To retrieve a specific file from Cloud Files:

```
/** @var $file OpenCloud\ObjectStore\Resource\DataObject */
$file = $container->getObject('summer_vacation.mp4');
```

Once you have access to this `OpenCloud\ObjectStore\Resource\DataObject` object, you can access these attributes:

### Get object's parent container

```
/** @param $container OpenCloud\ObjectStore\Resource\Container */
$container = $object->getContainer();
```

### Get file name

```
/** @param $name string */
$name = $object->getName();
```

### Get file size

```
/** @param $size int */
$size = $object->getContentLength();
```

### Get content of file

```
/** @param $content Guzzle\Http\EntityBody */
$content = $object->getContent();
```

### Get type of file

```
/** @param $type string */
$type = $object->getContentType();
```

### Get file checksum

```
/** @param $etag string */
$etag = $object->getEtag();
```

### Get last modified date of file

```
/** @param $lastModified string */
$lastModified = $object->getLastModified();
```

### Conditional requests

You can also perform conditional requests according to RFC 2616 specification (§§ 14.24-26). Supported headers are `If-Match`, `If-None-Match`, `If-Modified-Since` and `If-Unmodified-Since`.

So, to retrieve a file's contents only if it's been recently changed

```
$file = $container->getObject('error_log.txt', array(
    'If-Modified-Since' => 'Tue, 15 Nov 1994 08:12:31 GMT'
));

if ($file->getContentLength()) {
    echo 'Has been changed since the above date';
} else {
    echo 'Has not been changed';
}
```

Retrieve a file only if it has NOT been modified (and expect a 412 on failure):

```
use Guzzle\Http\Exception\ClientErrorResponseException;

try {
    $oldImmutableFile = $container->getObject('payroll_2001.xlsx', array(
        'If-Unmodified-Since' => 'Mon, 31 Dec 2001 23:00:00 GMT'
    ));
} catch (ClientErrorResponseException $e) {
    echo 'This file has been modified...';
}
```

Finally, you can specify a range - which will return a subset of bytes from the file specified. To return the last 20B of a file:

```
$snippet = $container->getObject('output.log', array('range' => 'bytes=-20'));
```

### Update an existing object

```
$file->setContent(fopen('/path/to/new/content', 'r+'));
$file->update();
```

Bear in mind that updating a file name will result in a new file being generated (under the new name). You will need to delete the old file.

### Copy object to new location

To copy a file to another location, you need to specify a string-based destination path:

```
$object->copy('/container_2/new_object_name');
```

Where `container_2` is the name of the container, and `new_object_name` is the name of the object inside the container that does not exist yet.

Get the executable PHP script for this example

### Symlinking to this object from another location

To create a symlink to this file in another location you need to specify a string-based source

```
$object->createSymlinkFrom('/container_2/new_object_name');
```

Where `container_2` is the name of the container, and `new_object_name` is the name of the object inside the container that either does not exist yet or is an empty file.

Get the executable PHP script for this example

### Setting this object to symlink to another location

To set this file to symlink to another location you need to specify a string-based destination

```
$object->createSymlinkTo('/container_2/new_object_name');
```

Where `container_2` is the name of the container, and `new_object_name` is the name of the object inside the container.

The object must be an empty file.

Get the executable PHP script for this example

### Get object metadata

You can fetch just the object metadata without fetching the full content:

```
$container->getPartialObject('summer_vacation.mp4');
```

In order to access the metadata on a partial or complete object, use:

```
$object->getMetadata();
```

You can turn a partial object into a full object to get the content after looking at the metadata:

```
$object->refresh();
```

You can also update to get the latest metadata:

```
$object->retrieveMetadata();
```

Get the executable PHP script for this example

### Update object metadata

Similarly, with setting metadata there are two options: you can update the metadata values of the local object (i.e. no HTTP request) if you anticipate you'll be executing one soon (an update operation for example):

```
// There's no need to execute a HTTP request, because we'll soon do one anyway for␣
↪the update operation
$object->setMetadata(array(
    'Author' => 'Hemingway'
));

// ... code here

$object->update();
```

Alternatively, you can update the API straight away - so that everything is retained:

```
$object->saveMetadata(array(
    'Author' => 'Hemingway'
));
```

Please be aware that these methods override and wipe existing values. If you want to append values to your metadata, use the correct method:

```
$metadata = $object->appendToMetadata(array(
  'Author' => 'Hemingway'
));

$object->saveMetadata($metadata);
```

Get the executable PHP script for this example

### Extract archive

CloudFiles provides you the ability to extract uploaded archives to particular destinations. The archive will be extracted and its contents will populate the particular area specified. To upload file (which might represent a directory structure) into a particular container:

```
use OpenCloud\ObjectStore\Constants\UrlType;

$service->bulkExtract('container_1', fopen('/home/jamie/files.tar.gz','r'),␣
↪UrlType::TAR_GZ);
```

You can also omit the container name (i.e. provide an empty string as the first argument). If you do this, the API will create the containers necessary to house the extracted files - this is done based on the filenames inside the archive.

Get the executable PHP script for this example

### Delete object

```
$container->deleteObject('{objectName}');
```

Get the executable PHP script for this example

### Delete already downloaded object

```
$object->delete();
```

Get the executable PHP script for this example

### Delete multiple objects

Bulk delete a set of paths:

```
$pathsToBeDeleted = array('/container_1/old_file', '/container_2/notes.txt', '/
↪container_1/older_file.log');

$service->batchDelete($pathsToBeDeleted);
```

Get the executable PHP script for this example

### Check an object exists

To check whether an object exists:

```
/** @var bool $exists */
$exists = $container->objectExists('{objectName}');
```

### CDN Containers

**Note:** This feature is only available to Rackspace users.

### Setup

In order to interact with CDN containers, you first need to instantiate a CDN service object:

```
$cdnService = $service->getCdnService();
```

### List CDN-enabled containers

To list CDN-only containers, follow the same operation for Storage which lists all containers. The only difference is which service object you execute the method on:

```
$cdnContainers = $cdnService->listContainers();

foreach ($cdnContainers as $cdnContainer) {
  /** @var $cdnContainer OpenCloud\ObjectStore\Resource\CDNContainer */
}
```

Get the executable PHP script for this example

### CDN-enable a container

Before a container can be CDN-enabled, it must exist in the storage system. When a container is CDN-enabled, any objects stored in it are publicly accessible over the Content Delivery Network by combining the container's CDN URL with the object name.

Any CDN-accessed objects are cached in the CDN for the specified amount of time called the TTL. The default TTL value is 259200 seconds, or 72 hours. Each time the object is accessed after the TTL expires, the CDN refetches and caches the object for the TTL period.

```
$container->enableCdn();
```

Get the executable PHP script for this example

### CDN-disable a container

```
$container->disableCdn();
```

Get the executable PHP script for this example

### Operations on CDN-enabled containers

Once a container has been CDN-enabled, you can retrieve it like so:

```
$cdnContainer = $cdnService->cdnContainer('{containerName}');
```

If you already have a container object and want to avoid instantiating a new service, you can also do:

```
$cdnContainer = $container->getCdn();
```

### Retrieve the SSL URL of a CDN container

```
$cdnContainer->getCdnSslUri();
```

### Retrieve the streaming URL of a CDN container

```
$cdnContainer->getCdnStreamingUri();
```

### Retrieve the iOS streaming URL of a CDN container

The Cloud Files CDN allows you to stream video to iOS devices without needing to convert your video. Once you CDN-enable your container, you have the tools necessary for streaming media to multiple devices.

```
$cdnContainer->getIosStreamingUri();
```

### CDN logging

To enable and disable logging for your CDN-enabled container:

```
$cdnContainer->enableCdnLogging();
$cdnContainer->disableCdnLogging();
```

### Purge CDN-enabled objects

To remove a CDN object from public access:

```
$object->purge();
```

You can also provide an optional e-mail address (or comma-delimeted list of e-mails), which the API will send a confirmation message to once the object has been completely purged:

```
$object->purge('jamie.hannaford@rackspace.com');
$object->purge('hello@example.com,hallo@example.com');
```

### Migrating containers across regions

Currently, there exists no single API operation to copy containers across geographic endpoints. Although the API offers a `COPY` operation for individual files, this does not work for cross-region copying. The SDK, however, does offer this functionality.

You **will** be charged for bandwidth between regions, so it's advisable to use ServiceNet where possible (which is free).

### Requirements

- You must install the full Guzzle package, so that the process can take advantage of Guzzle's batching functionality (it allows parallel requests to be batched for greater efficiency). You can do this by running:

```
composer require guzzle/guzzle
```

- Depending on the size and number of transfer items, you will need to raise PHP's memory limit:

```
ini_set('memory_limit', '512M');
```

- You will need to enact some kind of backoff/retry strategy for rate limits. Guzzle comes with a convenient feature that just needs to be added as a normal subscriber:

```
use Guzzle\Plugin\Backoff\BackoffPlugin;

// maximum number of retries
$maxRetries = 10;

// set HTTP error codes
$httpErrors = array(500, 503, 408);

$backoffPlugin = BackoffPlugin::getExponentialBackoff($maxRetries, $httpErrors);
$client->addSubscriber($backoffPlugin);
```

This tells the client to retry up to `10` times for failed requests have resulted in these HTTP status codes: `500`, `503` or `408`.

### Setup

You can access all this functionality by executing:

```
$ordService = $client->objectStoreService('cloudFiles', 'ORD');
$iadService = $client->objectStoreService('cloudFiles', 'IAD');

$oldContainer = $ordService->getContainer('old_container');
$newContainer = $iadService->getContainer('new_container');

$iadService->migrateContainer($oldContainer, $newContainer);
```

It's advisable to do this process in a Cloud Server in one of the two regions you're migrating to/from. This allows you to use `internalURL` as the third argument in the `objectStoreService` methods like this:

```
$client->objectStoreService('cloudFiles', 'IAD', 'internalURL');
```

This will ensure that traffic between your server and your new IAD container will be held over the internal Rackspace network which is free.

### Options

You can pass in an array of arguments to the method:

```
$options = array(
    'read.batchLimit'  => 100,
    'read.pageLimit'   => 100,
    'write.batchLimit' => 50
);

$iadService->migrateContainer($oldContainer, $newContainer, $options);
```

### Options explained

| Name | Description | Default |
|---|---|---|
| `read.pageLimit` | When the process begins, it has to collect all the files that exist in the old container. It does this through a conventional `objectList` method, which calls the `PaginatedIterator`. This iterator has the option to specify the page size for the collection (i.e. how many items are contained per page in responses from the API) | 10,000 |
| `read.batchLimit` | After the data objects are collected, the process needs to send an individual GET request to ascertain more information. In order to make this process faster, these individual GET requests are batched together and sent in parallel. This limit refers to how many of these GET requests are batched together. | 1,000 |
| `write.batchLimit` | Once each file has been retrieved from the API, a PUT request is executed against the new container. Similar to above, these PUT requests are batched - and this number refers to the amount of PUT requests batched together. | 100 |

### Temporary URLs

Temporary URLs allow you to create time-limited Internet addresses that allow you to grant access to your Cloud Files account. Using Temporary URL, you may allow others to retrieve or place objects in your containers - regardless of whether they're CDN-enabled.

### Set "temporary URL" metadata key

You must set this "secret" value on your account, where it can be used in a global state:

```
$account = $service->getAccount();
$account->setTempUrlSecret('my_secret');

echo $account->getTempUrlSecret();
```

The string argument of `setTempUrlSecret()` is optional - if left out, the SDK will generate a random hashed secret for you.

Get the executable PHP script for this example:

- Specify a URL secret
- Generate random URL secret

### Create a temporary URL

Once you've set an account secret, you can create a temporary URL for your object. To allow GET access to your object for 1 minute:

```
$object->getTemporaryUrl(60, 'GET');
```

To allow PUT access for 1 hour:

```
$object->getTemporaryUrl(360, 'PUT');
```

Get the executable PHP script for this example

### Override TempURL file names

Override tempURL file names simply by adding the filename parameter to the url:

```
$tempUrl = $object->getTemporaryUrl(60, 'GET');
$url = $tempUrl.'&filename='.$label;
```

### Hosting HTML sites on CDN

**Note:** This feature is only available to Rackspace users.

To host a static (i.e. HTML) website on Cloud Files, you must follow these steps:

1. CDN-enable a container:

```
$container = $service->getContainer('html_site');
$container->enableCdn();
```

2. Upload all HTML content. You can use nested directory structures.

```
$container->uploadObjects(array(
    array('name' => 'index.html', 'path' => 'index.html'),
    array('name' => 'contact.html', 'path' => 'contact.html'),
    array('name' => 'error.html', 'path' => 'error.html'),
    array('name' => 'styles.css', 'path' => 'styles.css'),
    array('name' => 'main.js', 'path' => 'main.js'),
));
```

3. Tell Cloud Files what to use for your default index page like this:

```
$container->setStaticIndexPage('index.html');
```

4. (Optional) Tell Cloud Files which error page to use by default:

```
$container->setStaticErrorPage('error.html');
```

Bear in mind that steps 3 & 4 do not upload content, but rather specify a reference to an existing page/CloudFiles object.

### Glossary

**account** The portion of the system designated for your use. An Object Store system is typically designed to be used by many different customers, and your user account is your portion of it.

**container** A storage compartment that provides a way for you to organize data. A container is similar to a folder in Windows or a directory in UNIX. The primary difference between a container and these other file system concepts is that containers cannot be nested.

**cdn** A system of distributed servers (network) that delivers web pages and other web content to a user based on the geographic locations of the user, the origin of the web page, and a content delivery server.

**metadata** Optional information that you can assign to Cloud Files accounts, containers, and objects through the use of a metadata header.

**object** An object (sometimes referred to as a file) is the unit of storage in an Object Store. An object is a combination of content (data) and metadata.

### Further links

- Getting Started Guide for the API
- API Developer Guide
- API release history

## Orchestration v1

### Setup

#### Rackspace setup

The first step is to pass in your credentials and set up a client. For Rackspace users, you will need your username and API key:

```
use OpenCloud\Rackspace;

$client = new Rackspace(Rackspace::US_IDENTITY_ENDPOINT, array(
  'username' => '{username}',
  'apiKey'   => '{apiKey}',
));
```

### OpenStack setup

If you're an OpenStack user, you will also need to prove a few other configuration parameters:

```
$client = new OpenCloud\OpenStack('{keystoneUrl}', array(
  'username' => '{username}',
  'password' => '{apiKey}',
  'tenantId' => '{tenantId}',
));
```

### Orchestration service

Now to instantiate the Orchestration service:

```
$service = $client->orchestrationService('{catalogName}', '{region}', '{urlType}');
```

- `{catalogName}` is the name of the service as it appears in the service catalog. OpenStack users *must* set this value. For Rackspace users, a default will be provided if you pass in `null`.
- `{region}` is the region the service will operate in. For Rackspace users, you can select one of the following from the *supported regions page*.
- `{urlType}` is the *type of URL* to use, depending on which endpoints your catalog provides. If omitted, it will default to the public network.

## Operations

### Templates

An Orchestration template is a JSON or YAML document that describes how a set of resources should be assembled to produce a working deployment (known as a *stack*). The template specifies the resources to use, the attributes of these resources that are parameterized and the information that is sent to the user when a template is instantiated.

### Validating templates

Before you use a template to create a stack, you might want to validate it.

### Validate a template from a file

If your template is stored on your local computer as a JSON or YAML file, you can validate it as shown in the following example:

```
use OpenCloud\Common\Exceptions\InvalidTemplateError;

try {
    $orchestrationService->validateTemplate(array(
        'template' => file_get_contents(__DIR__ . '/lamp.yaml')
    ));
} catch (InvalidTemplateError $e) {
    // Use $e->getMessage() for explanation of why template is invalid
}
```

Get the executable PHP script for this example

### Validate Template from URL

If your template is stored as a JSON or YAML file in a remote location accessible via HTTP or HTTPS, you can validate it as shown in the following example:

```
use OpenCloud\Common\Exceptions\InvalidTemplateError;

try {
    $orchestrationService->validateTemplate(array(
        'templateUrl' => 'https://raw.githubusercontent.com/rackspace-orchestration-
↪templates/lamp/master/lamp.yaml'
    ));
} catch (InvalidTemplateError $e) {
    // Use $e->getMessage() for explanation of why template is invalid
}
```

Get the executable PHP script for this example

### Stacks

A stack is a running instance of a template. When a stack is created, the *resources* specified in the template are created.

### Preview stack

Before you create a stack from a template, you might want to see what that stack will look like. This is called *previewing the stack*.

This operation takes one parameter, an associative array, with the following keys:

| Name | Description | Data type | Required? | Default value | Example value |
|---|---|---|---|---|---|
| `name` | Name of the stack | String. Must start with an alphabetic character, and must contain only alphanumeric, _, – or . characters | Yes | • | `simple-lamp-setup` |
| `template` | Template contents | String. JSON or YAML | No, if `templateUrl` is specified | `null` | `heat_template_version: 2013-05-23\ndescription: LAMP server\n` |
| `templateUrl` | URL of the template file | String. HTTP or HTTPS URL | No, if `template` is specified | `null` | `https://raw.githubusercontent.com/rackspace-orchestration-templates/lamp/master/lamp.yaml` |
| `parameters` | Arguments to the template, based on the template's parameters. For example, see the parameters in this template section | Associative array | No | `null` | `array('flavor_id' => 'general1-1')` |

### Preview a stack from a template file

If your template is stored on your local computer as a JSON or YAML file, you can use it to preview a stack as shown in the following example:

```
/** @var $stack OpenCloud\Orchestration\Resource\Stack **/
$stack = $orchestrationService->previewStack(array(
    'name'          => 'simple-lamp-setup',
    'template'      => file_get_contents(__DIR__ . '/lamp.yml'),
    'parameters'    => array(
        'server_hostname' => 'web01',
        'image'           => 'Ubuntu 14.04 LTS (Trusty Tahr) (PVHVM)'
    )
));
```

Get the executable PHP script for this example

### Preview a stack from a template URL

If your template is stored as a JSON or YAML file in a remote location accessible via HTTP or HTTPS, you can use it to preview a stack as shown in the following example:

```
/** @var $stack OpenCloud\Orchestration\Resource\Stack **/
$stack = $orchestrationService->previewStack(array(
    'name'         => 'simple-lamp-setup',
    'templateUrl'  => 'https://raw.githubusercontent.com/rackspace-orchestration-
↪templates/lamp/master/lamp.yaml',
    'parameters'   => array(
        'server_hostname' => 'web01',
        'image'           => 'Ubuntu 14.04 LTS (Trusty Tahr) (PVHVM)'
    )
));
```

Get the executable PHP script for this example

### Create stack

You can create a stack from a template. This operation takes one parameter, an associative array, with the following keys:

| Name | Description | Data type | Required? | Default value | Example value |
|---|---|---|---|---|---|
| `name` | Name of the stack | String. Must start with an alphabetic character, and must contain only alphanumeric, _, – or . characters. | Yes | • | `simple-lamp-setup` |
| `template` | Template contents | String. JSON or YAML | No, if `templateUrl` is specified | `null` | `heat_template_version: 2013-05-23\ndescription: LAMP server\n` |
| `templateUrl` | URL of template file | String. HTTP or HTTPS URL | No, if `template` is specified | `null` | `https://raw.githubusercontent.com/rackspace-orchestration-lamp/master/lamp.yaml` |
| `parameters` | Arguments to the template, based on the template's parameters | Associative array | No | `null` | `array('server_hostname' => 'web01')` |
| `timeoutMins` | Duration, in minutes, after which stack creation should time out | Integer | Yes | • | 5 |

### Create a stack from a template file

If your template is stored on your local computer as a JSON or YAML file, you can use it to create a stack as shown in the following example:

```
/** @var $stack OpenCloud\Orchestration\Resource\Stack **/
$stack = $orchestrationService->createStack(array(
    'name'         => 'simple-lamp-setup',
    'templateUrl'  => 'https://raw.githubusercontent.com/rackspace-orchestration-
↪templates/lamp/master/lamp.yaml',
    'parameters'   => array(
        'server_hostname' => 'web01',
        'image'           => 'Ubuntu 14.04 LTS (Trusty Tahr) (PVHVM)'
    ),
    'timeoutMins'  => 5
));
```

Get the executable PHP script for this example

### Create a stack from a template URL

If your template is stored as a JSON or YAML file in a remote location accessible via HTTP or HTTPS, you can use it to create a stack as shown in the following example:

```
$stack = $orchestrationService->stack();
$stack->create(array(
    'name'          => 'simple-lamp-setup',
    'templateUrl'   => 'https://raw.githubusercontent.com/rackspace-orchestration-
↪templates/lamp/master/lamp.yaml',
    'parameters'    => array(
        'server_hostname' => 'web01',
        'image'           => 'Ubuntu 14.04 LTS (Trusty Tahr) (PVHVM)'
    ),
    'timeoutMins'   => 5
));
```

Get the executable PHP script for this example

### List stacks

You can list all the stacks that you have created as shown in the following example:

```
$stacks = $orchestrationService->listStacks();
foreach ($stacks as $stack) {
    /** @var $stack OpenCloud\Orchestration\Resource\Stack **/
}
```

Get the executable PHP script for this example

### Get stack

You can retrieve a specific stack using its name, as shown in the following example:

```
/** @var $stack OpenCloud\Orchestration\Resource\Stack **/
$stack = $orchestrationService->getStack('simple-lamp-setup');
```

Get the executable PHP script for this example

### Get stack template

You can retrieve the template used to create a stack. Note that a JSON string is returned, regardless of whether a JSON or YAML template was used to create the stack.

```
/** @var $stackTemplate string **/
$stackTemplate = $stack->getTemplate();
```

Get the executable PHP script for this example

### Update stack

You can update a running stack.

This operation takes one parameter, an associative array, with the following keys:

| Name | Description | Data type | Required? | Default value | Example value |
|---|---|---|---|---|---|
| `template` | Template contents | String. JSON or YAML | No, if `templateUrl` is specified | `null` | `heat_template_version: 2013-05-23\ndescription: LAMP server\n` |
| `templateUrl` | URL of template file | String. HTTP or HTTPS URL | No, if `template` is specified | `null` | `https://raw.githubusercontent.com/rackspace-orchestration-templates/lamp/master/lamp-updated.yaml` |
| `parameters` | Arguments to the template, based on the template's parameters | Associative array | No | `null` | `array('flavor_id' => 'general1-1')` |
| `timeoutMins` | Duration, in minutes, after which stack update should time out | Integer | Yes | • | 5 |

### Update a stack from a template file

If your template is stored on your local computer as a JSON or YAML file, you can use it to update a stack as shown in the following example:

```
/** @var $stack OpenCloud\Orchestration\Resource\Stack **/
$stack->update(array(
    'template'      => file_get_contents(__DIR__ . '/lamp-updated.yml'),
    'parameters'    => array(
        'server_hostname' => 'web01',
        'image'           => 'Ubuntu 14.04 LTS (Trusty Tahr) (PVHVM)'
    ),
    'timeoutMins'   => 5
));
```

Get the executable PHP script for this example

### Update Stack from Template URL

If your template is stored as a JSON or YAML file in a remote location accessible via HTTP or HTTPS, you can use it to update a stack as shown in the following example:

```
/** @var $stack OpenCloud\Orchestration\Resource\Stack **/
$stack->update(array(
    'templateUrl'   => 'https://raw.githubusercontent.com/rackspace-orchestration-
↪templates/lamp/master/lamp-updated.yaml',
    'parameters'    => array(
        'server_hostname' => 'web01',
        'image'           => 'Ubuntu 14.04 LTS (Trusty Tahr) (PVHVM)'
    ),
    'timeoutMins'   => 5
));
```

Get the executable PHP script for this example

### Delete stack

If you no longer need a stack and all its resources, you can delete the stack *and* the resources as shown in the following example:

```
$stack->delete();
```

Get the executable PHP script for this example

### Abandon Stack

**Note:** This operation returns data about the abandoned stack as a string. You can use this data to recreate the stack by using the *adopt stack* operation.

If you want to delete a stack but preserve all its resources, you can abandon the stack as shown in the following example:

```
/** @var $abandonStackData string **/
$abandonStackData = $stack->abandon();
file_put_contents(__DIR__ . '/sample_adopt_stack_data.json', $abandonStackData);
```

Get the executable PHP script for this example

### Adopt stack

If you have data from an abandoned stack, you can re-create the stack as shown in the following example:

```
/** @var $stack OpenCloud\Orchestration\Resource\Stack **/
$stack = $orchestrationService->adoptStack(array(
    'name'           => 'simple-lamp-setup',
    'template'       => file_get_contents(__DIR__ . '/lamp.yml'),
    'adoptStackData' => $abandonStackData,
    'timeoutMins'    => 5
));
```

Get the executable PHP script for this example

### Stack resources

A stack is made up of zero or more resources such as databases, load balancers, and servers, and the software installed on servers.

### List stack resources

You can list all the resources for a stack as shown in the following example:

```
$resources = $stack->listResources();

foreach ($resources as $resource) {
    /** @var $resource OpenCloud\Orchestration\Resource\Resource **/
}
```

Get the executable PHP script for this example

### Get stack resource

You can retrieve a specific resource in a stack bt using that resource's name, as shown in the following example:

```
/** @var $resource OpenCloud\Orchestration\Resource\Resource **/
$resource = $stack->getResource('load-balancer');
```

Get the executable PHP script for this example

### Get stack resource metadata

You can retrieve the metadata for a specific resource in a stack as shown in the following example:

```
/** @var $resourceMetadata \stdClass **/
$resourceMetadata = $resource->getMetadata();
```

Get the executable PHP script for this example

### Resource types

When you define a template, you must use resource types supported by your cloud.

### List resource types

You can list all supported resource types as shown in the following example:

```
$resourceTypes = $orchestrationService->listResourceTypes();
foreach ($resourceTypes as $resourceType) {
    /** @var $resourceType OpenCloud\Orchestration\Resource\ResourceType **/
}
```

Get the executable PHP script for this example

### Get resource type

You can retrieve a specific resource type's schema as shown in the following example:

```
/** @var $resourceType OpenCloud\Orchestration\Resource\ResourceType **/
$resourceType = $orchestrationService->getResourceType('OS::Nova::Server');
```

Get the executable PHP script for this example

### Get resource type template

You can retrieve a specific resource type's representation as it would appear in a template, as shown in the following example:

```
/** @var $resourceTypeTemplate string **/
$resourceTypeTemplate = $resourceType->getTemplate();
```

Get the executable PHP script for this example

### Build info

### Get build info

You can retrieve information about the current Orchestration service build as shown in the following example:

```
/** @var $resourceType OpenCloud\Orchestration\Resource\BuildInfo **/
$buildInfo = $orchestrationService->getBuildInfo();
```

Get the executable PHP script for this example

### Stack resource events

Operations on resources within a stack (such as the creation of a resource) produce events.

### List stack events

You can list all of the events for all of the resources in a stack as shown in the following example:

```
$stackEvents = $stack->listEvents();

foreach ($stackEvents as $stackEvent) {
    /** @var $stackEvent OpenCloud\Orchestration\Resource\Event **/
}
```

Get the executable PHP script for this example

### List stack resource events

You can list all of the events for a specific resource in a stack as shown in the following example:

```
$resourceEvents = $resource->listEvents();

foreach ($resourceEvents as $resourceEvent) {
    /** @var $resourceEvent OpenCloud\Orchestration\Resource\Event **/
}
```

Get the executable PHP script for this example

### Get stack resource event

You can retrieve a specific event for a specific resource in a stack, by using the resource event's ID, as shown in the following example:

```
/** @var $resourceEvent OpenCloud\Orchestration\Resource\Event **/
$resourceEvent = $resource->getEvent('c1342a0a-59e6-4413-9af5-07c9cae7d729');
```

Get the executable PHP script for this example

## Glossary

**template** An Orchestration template is a JSON or YAML document that describes how a set of resources should be assembled to produce a working deployment. The template specifies what resources should be used, what attributes of these resources are parameterized and what information is output to the user when a template is instantiated.

**resource** A resource is a template artifact that represents some component of your desired architecture (a Cloud Server, a group of scaled Cloud Servers, a load balancer, some configuration management system, and so forth).

**stack** A stack is a running instance of a template. When a stack is created, the resources specified in the template are created.

## Further links

- Getting Started Guide for the API
- API Developer Guide
- API release history

# Queues v1

## Setup

### Rackspace setup

The first step is to pass in your credentials and set up a client. For Rackspace users, you will need your username and API key:

```
use OpenCloud\Rackspace;

$client = new Rackspace(Rackspace::US_IDENTITY_ENDPOINT, array(
  'username' => '{username}',
  'apiKey'   => '{apiKey}',
));
```

### OpenStack setup

If you're an OpenStack user, you will also need to prove a few other configuration parameters:

```
$client = new OpenCloud\OpenStack('{keystoneUrl}', array(
  'username' => '{username}',
  'password' => '{apiKey}',
  'tenantId' => '{tenantId}',
));
```

### Queues service

Now to instantiate the Queues service:

```
$service = $client->queuesService('{catalogName}', '{region}', '{urlType}');
```

- `{catalogName}` is the name of the service as it appears in the service catalog. OpenStack users *must* set this value. For Rackspace users, a default will be provided if you pass in `null`.
- `{region}` is the region the service will operate in. For Rackspace users, you can select one of the following from the *supported regions page*.
- `{urlType}` is the *type of URL* to use, depending on which endpoints your catalog provides. If omitted, it will default to the public network.

## Operations

### Queues

#### A note on Client IDs

For most of the operations in Cloud Queues, you must specify a **Client ID** which will be used as a unique identifier for the process accessing this Queue. This is basically a UUID that must be unique to each client accessing the API - it can be an arbitrary string.

```
$service->setClientId();

echo $service->getClientId();
```

If you call `setClientId` without any parameters, a UUID is automatically generated for you.

### List queues

This operation lists queues for the project. The queues are sorted alphabetically by name.

```
$queues = $service->listQueues();

foreach ($queues as $queue) {
    echo $queue->getName() , PHP_EOL;
}
```

### Filtering lists

You can also filter collections using the following query parameters:

```
$queues = $service->listQueues(array('detailed' => false));
```

### Create queue

The only parameter required is the name of the queue you're creating. The name must not exceed 64 bytes in length, and it is limited to US-ASCII letters, digits, underscores, and hyphens.

```
$queue = $service->createQueue('new_queue');
```

Get the executable PHP script for this example

### Find queue details

```
/** @var $queue OpenCloud\Queues\Resource\Queues */
$queue = $service->getQueue('{name}');
```

### Check queue existence

This operation verifies whether the specified queue exists by returning `TRUE` or `FALSE`.

```
if ($service->hasQueue('new_queue')) {
    // do something
}
```

### Update queue metadata

This operation replaces any existing metadata document in its entirety. Ensure that you do not accidentally overwrite existing metadata that you want to retain. If you want to *append* metadata, ensure you merge a new array to the existing values.

```
$queue->saveMetadata(array(
    'foo' => 'bar'
));
```

### Retrieve the queue metadata

This operation returns metadata, such as message TTL, for the queue.

```
$metadata = $queue->retrieveMetadata();
print_r($metadata->toArray());
```

### Get queue stats

This operation returns queue statistics, including how many messages are in the queue, categorized by status.

```
$queue->getStats();
```

### Delete queue

```
$queue->delete();
```

Get the executable PHP script for this example

### Messages

### Setup

In order to work with messages, you must first retrieve a queue by its name:

```
$queue = $service->getQueue('{queueName}');
```

### Post new message

This operation posts the specified message or messages. You can submit up to 10 messages in a single request.

When posting new messages, you specify only the `body` and `ttl` for the message. The API will insert metadata, such as ID and age.

How you pass through the array structure depends on whether you are executing multiple or single posts, but the keys are the same:

- The `body` attribute specifies an arbitrary document that constitutes the body of the message being sent. The size of this body is limited to 256 KB, excluding whitespace.

- The `ttl` attribute specifies how long the server waits before marking the message as expired and removing it from the queue. The value of ttl must be between 60 and 1209600 seconds (14 days). Note that the server might not actually delete the message until its age has reached up to (ttl + 60) seconds, to allow for flexibility in storage implementations.

### Posting a single message

```
use OpenCloud\Common\Constants\Datetime;

$queue->createMessage(array(
    'body' => (object) array(
        'event'    => 'BackupStarted',
        'deadline' => '26.12.2013',
    ),
    'ttl'  => 2 * Datetime::DAY
));
```

Get the executable PHP script for this example

### Post a batch of messages

Please note that the list of messages will be truncated at 10. For more, please execute another method call.

```
use OpenCloud\Common\Constants\Datetime;

$messages = array(
    array(
        'body' => (object) array(
            'play' => 'football'
        ),
        'ttl'  => 2 * Datetime::DAY
    ),
    array(
        'body' => (object) array(
            'play' => 'tennis'
        ),
        'ttl'  => 50 * Datetime::HOUR
    )
);

$queue->createMessages($messages);
```

### Get messages

This operation gets the message or messages in the specified queue.

Message IDs and markers are opaque strings. Clients should make no assumptions about their format or length. Furthermore, clients should assume that there is no relationship between markers and message IDs (that is, one cannot be derived from the other). This allows for a wide variety of storage driver implementations.

Results are ordered by age, oldest message first.

### Parameters

When retrieving messages, you can filter using these options:

```
$messages = $queue->listMessages(array(
    'marker' => '51db6f78c508f17ddc924357',
    'limit'  => 20,
    'echo'   => true
));

foreach ($messages as $message) {
    echo $message->getId() . PHP_EOL;
}
```

### Get a set of messages by ID

This operation provides a more efficient way to query multiple messages compared to using a series of individual GET. Note that the list of IDs cannot exceed 20. If a malformed ID or a nonexistent message ID is provided, it is ignored, and the remaining messages are returned.

### Parameters

```
$ids = array('id_1', 'id_2');

$messages = $queue->listMessages(array('ids' => $ids));

foreach ($messages as $message) {
    echo $message->getId() . PHP_EOL;
}
```

### Delete a set of messages by ID

This operation immediately deletes the specified messages. If any of the message IDs are malformed or non-existent, they are ignored. The remaining valid messages IDs are deleted.

```
$ids = array('id_1', 'id_2');
$response = $queue->deleteMessages($ids);
```

### Get a specific message

This operation gets the specified message from the specified queue.

```
/** @var $message OpenCloud\Queues\Message */
$message = $queue->getMessage('{messageId}');
```

Once you have access to the `Message` object, you access its attributes:

| attribute | method | description |
|---|---|---|
| href | `getHref` | An opaque relative URI that the client can use to uniquely identify a message resource and interact with it. |
| ttl | `getTtl` | The TTL that was set on the message when it was posted. The message expires after (ttl - age) seconds. |
| age | `getAge` | The number of seconds relative to the server's clock. |
| body | `getBody` | The arbitrary document that was submitted with the original request to post the message. |

### Delete message

```
$message->delete();
```

### Claims

### Setup

In order to work with messages, you must first retrieve a queue by its name:

```
$queue = $service->getQueue('{queueName}');
```

### Claim messages

This operation claims a set of messages (up to the value of the limit parameter) from oldest to newest and skips any messages that are already claimed. If no unclaimed messages are available, the API returns a `204 No Content` message.

When a client (worker) finishes processing a message, it should delete the message before the claim expires to ensure that the message is processed only once. As part of the delete operation, workers should specify the claim ID (which is best done by simply using the provided href). If workers perform these actions, then if a claim simply expires, the server can return an error and notify the worker of the race condition. This action gives the worker a chance to roll back its own processing of the given message because another worker can claim the message and process it.

The age given for a claim is relative to the server's clock. The claim's age is useful for determining how quickly messages are getting processed and whether a given message's claim is about to expire.

When a claim expires, it is released. If the original worker failed to process the message, another client worker can then claim the message.

### Parameters

The `ttl` attribute specifies how long the server waits before releasing the claim. The ttl value must be between 60 and 43200 seconds (12 hours). You must include a value for this attribute in your request.

The `grace` attribute specifies the message grace period in seconds. The value of grace value must be between 60 and 43200 seconds (12 hours). You must include a value for this attribute in your request. To deal with workers that have stopped responding (for up to 1209600 seconds or 14 days, including claim lifetime), the server extends the lifetime of claimed messages to be at least as long as the lifetime of the claim itself, plus the specified grace period. If a claimed message would normally live longer than the grace period, its expiration is not adjusted.

The `limit` attribute specifies the number of messages to return, up to 20 messages. If limit is not specified, limit defaults to 10. The limit parameter is optional.

```
use OpenCloud\Common\Constants\Datetime;

$queue->claimMessages(array(
    'limit' => 15,
    'grace' => 5 * Datetime::MINUTE,
    'ttl'   => 5 * Datetime::MINUTE
));
```

Get the executable PHP script for this example

### Query claim

This operation queries the specified claim for the specified queue. Claims with malformed IDs or claims that are not found by ID are ignored.

```
$claim = $queue->getClaim('{claimId}');
```

### Update claim

This operation updates the specified claim for the specified queue. Claims with malformed IDs or claims that are not found by ID are ignored.

Clients should periodically renew claims during long-running batches of work to avoid losing a claim while processing a message. The client can renew a claim by executing this method on a specific **Claim** and including a new TTL. The API will then reset the age of the claim and apply the new TTL.

```
use OpenCloud\Common\Constants\Datetime;

$claim->update(array(
    'ttl' => 10 * Datetime::MINUTE
));
```

### Release claim

This operation immediately releases a claim, making any remaining undeleted messages that are associated with the claim available to other workers. Claims with malformed IDs or claims that are not found by ID are ignored.

This operation is useful when a worker is performing a graceful shutdown, fails to process one or more messages, or is taking longer than expected to process messages, and wants to make the remainder of the messages available to other workers.

```
$message->delete();
```

## Glossary

**claim** A Claim is the process of a worker checking out a message to perform a task. Claiming a message prevents other workers from attempting to process the same messages.

**queue** A Queue is an entity that holds messages. Ideally, a queue is created per work type. For example, if you want to compress files, you would create a queue dedicated to this job. Any application that reads from this queue would only compress files.

**message** A Message is a task, a notification, or any meaningful data that a producer or publisher sends to the queue. A message exists until it is deleted by a recipient or automatically by the system based on a TTL (time-to-live) value.

### Further links

- Getting Started Guide for the API
- API Developer Guide
- API release history

# Volumes v1

## Setup

### Rackspace setup

The first step is to pass in your credentials and set up a client. For Rackspace users, you will need your username and API key:

```
use OpenCloud\Rackspace;

$client = new Rackspace(Rackspace::US_IDENTITY_ENDPOINT, array(
  'username' => '{username}',
  'apiKey'   => '{apiKey}',
));
```

### OpenStack setup

If you're an OpenStack user, you will also need to prove a few other configuration parameters:

```
$client = new OpenCloud\OpenStack('{keystoneUrl}', array(
  'username' => '{username}',
  'password' => '{apiKey}',
  'tenantId' => '{tenantId}',
));
```

### Volume service

Now to instantiate the Volume service:

```
$service = $client->volumeService('{catalogName}', '{region}', '{urlType}');
```

- `{catalogName}` is the name of the service as it appears in the service catalog. OpenStack users *must* set this value. For Rackspace users, a default will be provided if you pass in `null`.

- `{region}` is the region the service will operate in. For Rackspace users, you can select one of the following from the *supported regions page*.
- `{urlType}` is the *type of URL* to use, depending on which endpoints your catalog provides. If omitted, it will default to the public network.

## Operations

### Volumes

### Create a volume

To create a volume, you must specify its size (in gigabytes). All other parameters are optional:

```
// Create instance of OpenCloud\Volume\Resource\Volume
$volume = $service->volume();

$volume->create(array(
    'size'                => 200,
    'volume_type'         => $service->volumeType('<volume_type_id>'),
    'display_name'        => 'My Volume',
    'display_description' => 'Used for large object storage'
));
```

Get the executable PHP script for this example

### List volumes

```
$volumes = $service->volumeList();

foreach ($volumes as $volume) {
    /** @param $volumeType OpenCloud\Volume\Resource\Volume */
}
```

Get the executable PHP script for this example

### Get details on a single volume

If you specify an ID on the `volume()` method, it retrieves information on the specified volume:

```
$volume = $dallas->volume('<volume_id>');
echo $volume->size;
```

Get the executable PHP script for this example

### To delete a volume

```
$volume->delete();
```

Get the executable PHP script for this example

### Attach a volume to a server

```
// retrieve server
$computeService = $client->computeService('{catalogName}', '{region}');
$server = $computeService->server('{serverId}');

// attach volume
$server->attachVolume($volume, '{mountPoint}')
```

The `{mountPoint}` is the location on the server on which to mount the volume (usually `/dev/xvhdd` or similar). You can also supply `'auto'` as the mount point, in which case the mount point will be automatically selected for you. `auto` is the default value for `{mountPoint}`, so you do not actually need to supply anything for that parameter.

### Detach a volume from a server

```
$server->detachVolume($volume);
```

### Volume Types

### List volume types

```
$volumeTypes = $service->volumeTypeList();

foreach ($volumeTypes as $volumeType) {
    /** @param $volumeType OpenCloud\Volume\Resource\VolumeType */
}
```

### Describe a volume type

If you know the ID of a volume type, use the `volumeType` method to retrieve information on it:

```
$volumeType = $service->volumeType(1);
```

A volume type has three attributes:

- `id` the volume type identifier
- `name` its name
- `extra_specs` additional information for the provider

### Snapshots

### Create a snapshot

A `Snapshot` object is created from the Cloud Block Storage service. However, it is associated with a volume, and you must specify a volume to create one:

```
// New instance of OpenCloud\Volume\Resource\Snapshot
$snapshot = $service->snapshot();

// Send to API
$snapshot->create(array(
    'display_name' => 'Name that snapshot',
    'volume_id'    => $volume->id
));
```

Get the executable PHP script for this example

### List snapshots

```
$snapshots = $service->snapshotList();

foreach ($snapshots as $snapshot) {
    /** @param $snapshot OpenCloud\Volume\Resource\Snapshot */
}
```

Get the executable PHP script for this example

### To get details on a single snapshot

```
$snapshot = $dallas->snapshot('{snapshotId}');
```

Get the executable PHP script for this example

### To delete a snapshot

```
$snapshot->delete();
```

Get the executable PHP script for this example

## Glossary

**volume** A volume is a detachable block storage device. You can think of it as a USB hard drive. It can only be attached to one instance at a time.

**volume type** Providers may support multiple types of volumes; at Rackspace, a volume can either be `SSD` (solid state disk: expensive, high-performance) or `SATA` (serial attached storage: regular disks, less expensive).

**snapshot** A snapshot is a point-in-time copy of the data contained in a volume.

## Further links

- Getting Started Guide for the API
- API Developer Guide
- API release history

CHAPTER 3

# Usage tips

## Debugging

There are two important debugging strategies to use when encountering problems with HTTP transactions.

### Strategy 1: Meaningful exception handling

If the API returns a `4xx` or `5xx` status code, it indicates that there was an error with the sent request, meaning that the transaction cannot be adequately completed.

The Guzzle HTTP component, which forms the basis of our SDK's transport layer, utilizes numerous exception classes to handle this error logic.

The two most common exception classes are:

- `Guzzle\Http\Exception\ClientErrorResponseException`, which is thrown when a `4xx` response occurs
- `Guzzle\Http\Exception\ServerErrorResponseException`, which is thrown when a `5xx` response occurs

Both of these classes extend the base `BadResponseException` class.

This provides you with the granularity you need to debug and handle exceptions.

#### An example with Swift

If you're trying to retrieve a Swift resource, such as a Data Object, and you're not completely certain that it exists, it makes sense to wrap your call in a try/catch block:

```
use Guzzle\Http\Exception\ClientErrorResponseException;

try {
    return $service->getObject('foo.jpg');
```

```
} catch (ClientErrorResponseException $e) {
    if ($e->getResponse()->getStatusCode() == 404) {
      // Okay, the resource does not exist
      return false;
    }
} catch (\Exception $e) {
    // Some other exception was thrown...
}
```

Both `ClientErrorResponseException` and `ServerErrorResponseException` have two methods that allow you to access the HTTP transaction:

```
// Find out the faulty request
$request = $e->getRequest();

// Display everything by casting as string
echo (string) $request;

// Find out the HTTP response
$response = $e->getResponse();

// Output that too
echo (string) $response;
```

### Strategy 2: Wire logging

Guzzle provides a Log plugin that allows you to log everything over the wire, which is useful if you don't know what's going on.

Here's how you enable it:

#### Install the plugin

```
composer require guzzle/guzzle
```

#### Add to your client

```
use Guzzle\Plugin\Log\LogPlugin;

$client->addSubscriber(LogPlugin::getDebugPlugin());
```

The above will add a generic logging subscriber to your client, which will output every HTTP transaction to *STDOUT*.

## Caching credentials

You can speed up your API operations by caching your credentials in a (semi-)permanent location, such as your DB or local filesystem. This enable subsequent requests to access a shared resource, instead of repetitively having to re-authenticate on every thread of execution.

Tokens are valid for 24 hours, so you can effectively re-use the same cached value for that period. If you try to use a cached version that has expired, an authentication request will be made.

### Filesystem example

In this example, credentials will be saved to a file in the local filesystem. Be sure to exclude it from your VCS.

```
use OpenCloud\Rackspace;

$client = new Rackspace(Rackspace::US_IDENTITY_ENDPOINT, array(
    'username' => 'foo',
    'apiKey'   => 'bar'
));

$cacheFile = __DIR__ . '/.opencloud_token';

// If the cache file exists, try importing it into the client
if (file_exists($cacheFile)) {
    $data = unserialize(file_get_contents($cacheFile));
    $client->importCredentials($data);
}

$token = $client->getTokenObject();

// If no token exists, or the current token is expired, re-authenticate and save the␣
↪new token to disk
if (!$token || ($token && $token->hasExpired())) {
    $client->authenticate();
    file_put_contents($cacheFile, serialize($client->exportCredentials()));
}
```

In tests, the above code shaved about 1-2s off the execution time.

## Iterators

Iterators allow you to traverse over collections of your resources in an efficient and easy way. Currently there are two Iterators provided by the SDK:

- **ResourceIterator**. The standard iterator class that implements SPL's standard Iterator, ArrayAccess and Countable interfaces. In short, this allows you to traverse this object (using `foreach`), count its internal elements like an array (using `count` or `sizeof`), and access its internal elements like an array (using `$iterator[1]`).
- **PaginatedIterator**. This is a child of ResourceIterator, and as such inherits all of its functionality. The difference however is that when it reaches the end of the current collection, it attempts to construct a URL to access the API based on predictive paginated collection templates.

### Common behaviour

```
$iterator = $computeService->flavorList();
```

There are two ways to traverse an iterator. The first is the longer, more traditional way:

```
while ($iterator->valid()) {
    $flavor = $iterator->current();

    // do stuff..
    echo $flavor->id;
```

```
    $iterator->next();
}
```

There is also a shorter and more intuitive version:

```
foreach ($iterator as $flavor) {
    // do stuff...
    echo $flavor->id;
}
```

Because the iterator implements PHP's native `Iterator` interface, it can inherit all the native functionality of traversible data structures with `foreach`.

## Very important note

Until now, users have been expected to do this:

```
while ($flavor = $iterator->next()) {
   // ...
}
```

which is **incorrect**. The single responsibility of `next` is to move the internal pointer forward. It is the job of `current` to retrieve the current element.

For your convenience, these two Iterator classes are fully backward compatible: they exhibit all the functionality you'd expect from a correctly implemented iterator, but they also allow previous behaviour.

## Using paginated collections

For large collections, such as retrieving DataObjects from CloudFiles/Swift, you need to use pagination. Each resource will have a different limit per page; so once that page is traversed, there needs to be another API call to retrieve to *next* page's resources.

There are two key concepts:

- **limit** is the amount of resources returned per page
- **marker** is the way you define a starting point. It is some form of identifier that allows the collection to begin from a specific resource

### Resource classes

When the iterator returns a current element in the internal list, it populates the relevant resource class with all the data returned to the API. In most cases, a `stdClass` object will become an instance of `OpenCloud\Common\PersistentObject`.

In order for this instantiation to happen, the `resourceClass` option must correspond to some method in the parent class that creates the resource. For example, if we specify 'ScalingPolicy' as the `resourceClass`, the parent object (in this case `OpenCloud\Autoscale\Group`, needs to have some method will allows the iterator to instantiate the child resource class. These are all valid:

1. `Group::scalingGroup($data);`
2. `Group::getScalingGroup($data);`
3. `Group::resource('ScalingGroup', $data);`

where `$data` is the standard object. This list runs in order of precedence.

## Setting up a PaginatedIterator

```
use OpenCloud\Common\Collection\PaginatedIterator;

$service = $client->computeService();

$flavors = PaginatedIterator::factory($service, array(
    'resourceClass'  => 'Flavor',
    'baseUrl'        => $service->getUrl('flavors')
    'limit.total'    => 350,
    'limit.page'     => 100,
    'key.collection' => 'flavors'
));

foreach ($flavors as $flavor) {
    echo $flavor->getId();
}
```

As you can see, there are a lot of configuration parameters to pass in - and getting it right can be quite fiddly, involving a lot of API research. For this reason, using the convenience methods like `flavorList` is recommended because it hides the complexity.

### PaginatedIterator options

There are certain configuration options that the paginated iterator needs to work. These are:

| Name | Description | Type | Required | Default |
|---|---|---|---|---|
| resourceClass | The resource class that is instantiated when the current element is retrieved. This is relative to the parent/service which called the iterator. | string | Yes | • |
| baseUrl | The base URL that is used for making new calls to the API for new pages | `Guzzle\Http\Url` | Yes | • |
| limit.total | The total amount of resources you want to traverse in your collection. The iterator will stop as this limit is reached, regardless if there are more items in the list | int | No | 10000 |
| limit.page | The amount of resources each page contains | int | No | 100 |
| key.links | Often, API responses will contain "links" that allow easy access to the next page of a resource collection. This option specifies what that JSON element is called (its key). For example, for Rackspace Compute images it is `images_links`. | string | No | links |
| key.collection | The top-level key for the array of resources. For example, servers are returned with this data structure: `{"servers": [...]}`. The **key.collection** value in this case would be `servers`. | string | No | `null` |
| key.collectionElement | Rarely used. But it indicates the key name for each nested resource element. KeyPairs, for example, are listed like this: `{"keypairs": [ {"keypair": {...}} ] }`. So in this case | string | No | `null` |

# Rackspace regions

Below are the supported regions on the Rackspace network:

| code | location |
|---|---|
| IAD | Virginia |
| ORD | Chicago |
| DFW | Dallas |
| LON | London |
| SYD | Sydney |
| HKG | Hong Kong |

# URL types

## internalURL

An internal URL is a URL that is accessible only from within the Rackspace Cloud network. Access to an internal URL is always free of charge.

## publicURL

A public URL is a URL that is accessible from anywhere. Access to a public URL usually incurs traffic charges.

# Logging

## Logger injection

As the `Rackspace` client extends the `OpenStack` client, they both support passing `$options` as an array via the constructor's third parameter. The options are passed as a config to the *Guzzle* client, but also allow to inject your own logger.

Your logger should implement the `Psr\Log\LoggerInterface` as defined in PSR-3. One example of a compatible logger is Monolog. When the client does create a service, it will inject the logger if one is available.

To inject a `LoggerInterface` compatible logger into a new client:

```
use Monolog\Logger;
use OpenCloud\OpenStack;

// create a log channel
$logger = new Logger('name');

$client = new OpenStack('http://identity.my-openstack.com/v2.0', array(
  'username' => 'foo',
  'password' => 'bar'
), array(
  'logger' => $logger,
));
```

# HTTP Clients

## Default HTTP headers

To set default HTTP headers:

```
$client->setDefaultOption('headers/X-Custom-Header', 'FooBar');
```

## User agents

php-opencloud will send a default `User-Agent` header for every HTTP request, unless a custom value is provided by the end-user. The default header will be in this format:

> User-Agent: OpenCloud/xxx cURL/yyy PHP/zzz

where `xxx` is the current version of the SDK, `yyy` is the current version of cURL, and `zzz` is the current PHP version. To override this default, you must run:

```
$client->setUserAgent('MyCustomUserAgent');
```

which will result in:

> User-Agent: MyCustomUserAgent

If you want to set a *prefix* for the user agent, but retain the default `User-Agent` as a suffix, you must run:

```
$client->setUserAgent('MyPrefix', true);
```

which will result in:

> User-Agent: MyPrefix OpenCloud/xxx cURL/yyy PHP/zzz

where `$client` is an instance of `OpenCloud\OpenStack` or `OpenCloud\Rackspace`.

## Other functionality

For a full list of functionality provided by Guzzle, please consult the official documentation.

# Authentication

The client does not automatically authenticate against the API when it is instantiated - it waits for an API call. When this happens, it checks whether the current "token" has expired, and (re-)authenticates if necessary.

You can force authentication, by calling:

```
$client->authenticate();
```

If the credentials are incorrect, a `401` error will be returned. If credentials are correct, a `200` status is returned with your Service Catalog.

## Service Catalog

The Service Catalog is returned on successful authentication, and is composed of all the different API services available to the current tenant. All of this functionality is encapsulated in the `Catalog` object, which allows you greater control and interactivity.

```
/** @var OpenCloud\Common\Service\Catalog */
$catalog = $client->getCatalog();

// Return a list of OpenCloud\Common\Service\CatalogItem objects
foreach ($catalog->getItems() as $catalogItem) {

    $name = $catalogItem->getName();
    $type = $catalogItem->getType();

    if ($name == 'cloudServersOpenStack' && $type == 'compute') {
        break;
    }

    // Array of OpenCloud\Common\Service\Endpoint objects
    $endpoints = $catalogItem->getEndpoints();
    foreach ($endpoints as $endpoint) {
        if ($endpoint->getRegion() == 'DFW') {
            echo $endpoint->getPublicUrl();
        }
    }
}
```

As you can see, you have access to each Service's name, type and list of endpoints. Each endpoint provides access to the specific region, along with its public and private endpoint URLs.

CHAPTER 4

# Help and support

If you have specific problems or bugs with this SDK, please file an issue on our official Github. We also have a mailing list, so feel free to join to keep up to date with all the latest changes and announcements to the library.

You can also find assistance via IRC on #rackspace at freenode.net.

CHAPTER 5

# Contributing

If you'd like to contribute to the project, or require help running the unit/acceptance tests, please view the contributing guidelines.

# Index

## W