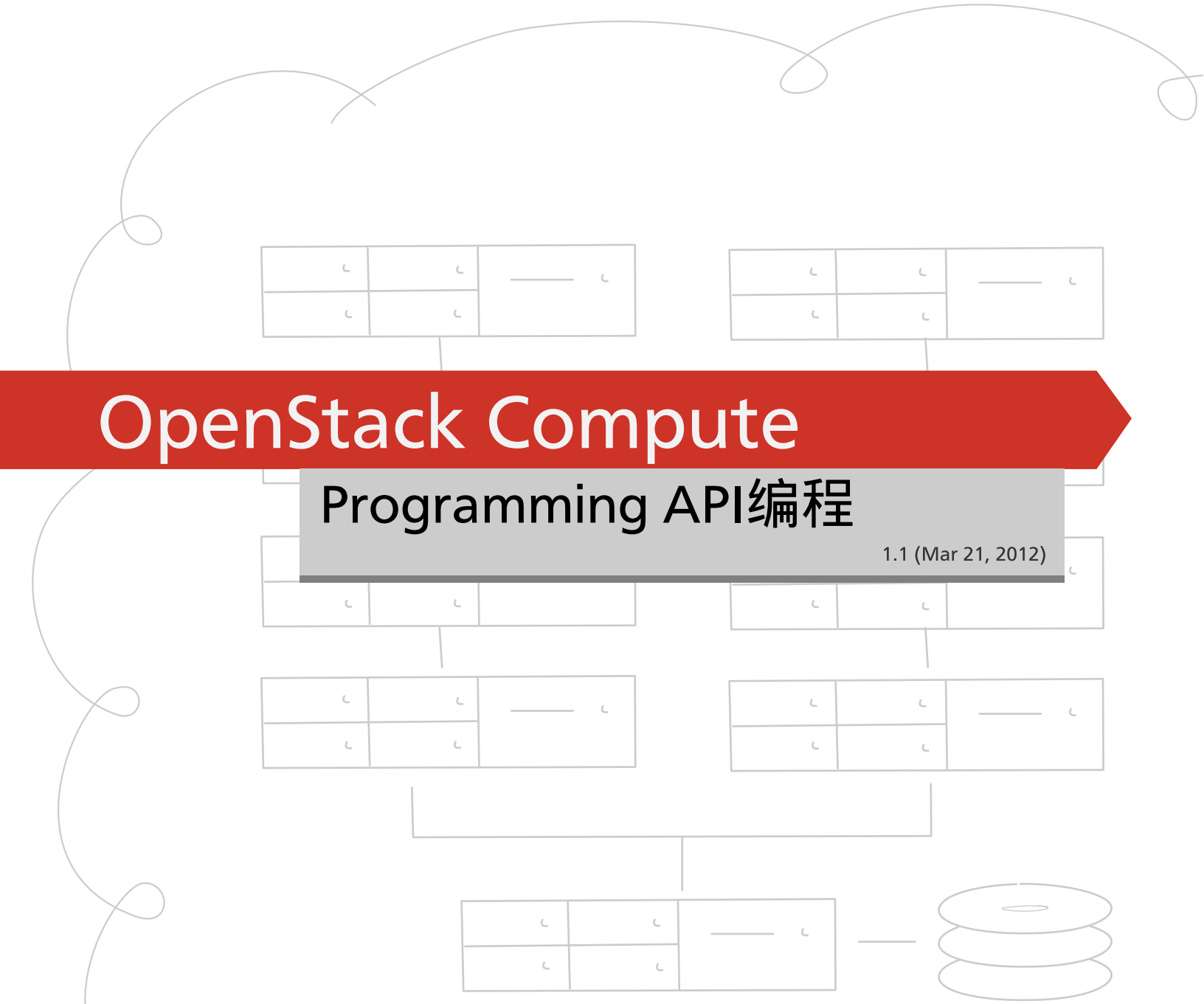
OpenStack Compute

Programming API编程

1.1 (Mar 21, 2012)



openstack™

docs.openstack.org

Programming OpenStack Compute API编程

Jacek Artymiak

1.1 (2012-03-21)

摘要

This is a book about programming the OpenStack Compute Nova API, v1.1. It is meant to be read by both programmers and system administrators familiar with programming the Unix shell (Bash, sh) or writing code in the Python programming language. As such it will be of interest to any system administrator and programmer using BSD, Linux, Unix, Mac OS X, and Microsoft Windows operating systems.

目录

第 1 章 Why Should You Use Cloud Computing?

Cloud computing is a way to build flexible server farms that can augment or replace the old-style server farms with their high maintenance cost and inflexible hardware configuration. They make it possible to do the things that are impossible with traditional hardware.

Thanks to cloud computing, things like replacing Microsoft Windows Server with Linux on 100+ machines; replacing 10 1-core servers with 10 4-core servers; or, doubling the number of the front-end servers to handle the periodic increase in traffic and then taking them off-line when traffic drops off, can be done in a couple of minutes. And since you can rent cloud servers by the hour, you could boot 100 of them to do in one hour a data processing job that would otherwise take one server 100 hours to complete. The price for that job will be the same. Try doing that with your traditional data center! (And don't forget to compare the pricing.)

How Will You Manage It All?

One of the first things you may think of is how on Earth are you supposed to manage all those computing and storage resources? All cloud computing services providers offer basic web-based control panels, which are good enough as general administration dashboards, but most probably won't meet your needs, if you want to create a dynamically reconfigurable virtual server farm.

A better way to manage your cloud is via a set of scripts that communicate with your cloud provider's infrastructure. This is done via a Programming Application Interface (API), which is offered by virtually any cloud computing provider.

The problem with APIs is that there are so many of them. Every cloud provider's infrastructure is different and that fact is reflected in the functionality exposedvia their APIs. Although the fragmentation of the cloud management APIs is genuinely justified, it doesn't make any easier for cloud server farm administrators to deploy their servers on different cloud infrastructures.

The Solution: OpenStack Compute API

Fortunately, you are not alone with this problem. The fanatically helpful people at Rackspace have decided that it would be beneficial for their customers and themselves if they created an open standard for common cloud services along with an accompanying open cloud management API. That was the beginning of the OpenStack project and the OpenStack Compute API. Being open, it can be used by other hosting companies.

When other cloud computing providers adopt the OpenStack Compute API, you will be able to deploy your servers with different providers using the same set of tools by changing only three pieces of information in your code:

- the URL of the endpoint for the calls to the API;
- the username of the authorized user;

- the password of the authorized user.

From the programmer's point of view, OpenStack Compute API looks like another web API, similar to the Flickr API or the Twitter API—if someone decides to clone the Flickr API you will not have to change anything in your own code. Similarly, if your cloud infrastructure provider offers OpenStack Compute cloud servers, storage, and APIs, you do not have to learn the new ways of doing similar things, you just change the endpoint URL and your user credentials.

第 2 章 The Basics

There are two ways to start learning OpenStack Compute API programming—you can get an account with an OpenStack cloud infrastructure provider or you can download a test virtual machine environment from the OpenStack project's servers by following the instructions published at this URL:

- http://wiki.openstack.org/SingleNodeNovaVagrantChef

Please bear in mind that the OpenStack project is moving at a fairly rapid pace and things sometimes don't work or need to be fixed by hand. For example, the trainig environment configuration may have to be adjusted a little. When you hit a snag, ask for help on the OpenStack site.

- http://openstack.org

Sometimes you won't have to ask. The OpenStack project is well-documented and you can find all of the necessary information at the following URL:

- http://docs.openstack.org

Getting Python

The OpenStack Compute Nova API can be accessed in any programming language. In this book we present examples written in the Python programming languge and if you want to follow them on your own, you will need a Python interpreter.

Ideally, you should be using the latest Python interpreter from the 2.x branch, which can be any version starting with number 2.6.1 upwards. The Python interpreter does not have to be installed on you virtual server machines, just the ones you will be using to communicate with the OpenStack Compute Application Programming Interface (API).

Let's walk through the installation process.

Installing Python on Your Machine

Python is a popular scripting language that ships pre-installed on Mac OS X and some Linux distributions. You can verify if and which version of the Python interpreter you have installed on your system with the following command (must be issued on the command line, do not type the dollar sign):

```
$ python --version
```

The version you need is at least 2.6.1. (Please use the latest 2.x.x version.)

If you are working in a graphical user interface environment and you are not sure how to get to the command line, try the following advice:

- Linux — click on the toolbar located along one of the edges of the screen and locate the Terminal menu option in the Accessories menu. On Ubuntu Linux you will find it in Applications > Accessories > Terminal.

- Mac OS X — press Ctrl+Space to display the Spotlight search field. Type terminal and select the Terminal application from the list of results.
- Microsoft Windows — click the Start menu button, select Run, type cmd and hit the Enter/Return key.

If you do not have Python installed on your system or it is a version earlier than 2.6.1, you will need to add a Python interpreter to your system by hand.

Python installation packages exist for all popular operating systems, and if you cannot get one for your favorite work environment, you can usually build it from the sources. You can download the Python interpreter binaries or sources from the official Python download page:

http://www.python.org/getit/

If your favorite operating system is not listed on that page, refer to your system's documentation for information on adding external packages. Various variants of Unix have their own packaging systems that make sure the software you install is properly configured on your system. For example, the right way to add Python on Ubuntu Linux is via the apt-get command:

```
$ sudo apt-get install python
```

Whenever possible, use pre-made Python 2.x packages. You can always build software from the sources, but it should be your last choice, if you are looking for convenience.

If you are installing Python on a computer running Microsoft Windows, you may have to add the path to the Python interpreter to the Path environment variable. On a Microsoft Windows 7 system, you will need to open the Environment Variables dialog. You can do it in the following way:

1. Click the Start button.
2. Right-click the Computer menu item.
3. Select Properties.
4. Select Advanced System Settings.
5. Click the Environment Variables button.
6. Click the Path entry on the System Variables list.
7. Click the Edit button.
8. Add ;C:26 (or ;C:27 for Python 2.7.x) to the end of the string shown in the text field of the edit dialog.
9. Click the OK button.

Please check the real path to the Python interpreter on your system. You can do that by browsing the system disk. Look for the topmost folder/path whose name begins with Python.

Examples

Examples from this book can be found in the repository found at the following URL:

- http://devguide.net/books/openstackapi

Please note that all examples in this book have been written in a way that is meant to explain the concepts and the interactions between your code and the OpenStack Compute API. In real life, you will want to use additional error handling and data processing code.

Getting the Keys to the Kingdom

Depending on the setup you are going to use, there could be three ways to authenticate yourself with an OpenStack Compute v1.1 API:

1. Username/password — needs to be given with every call to the OpenStack Compute API. Not very convenient nor secure.

2. An authentication token — typically OAuth, but could be any kind of a secret string of bits with an expiry date and time. This is the preferred way to authenticate yourself when calling the OpenStack Compute API. You obtain your authentication token from the authentication server using your username and password, which has to be given only once during the lifetime of the token.

3. An API key — works just like the authentication token, but does not have an expiry date/time. You or your OpenStack provider's administrator/support staff can revoke it and generate a new one. It is not as safe as an authentication token, which can have a short lifespan, but at least you don't have to send your username and password with every API call. You don't have to send those credentials at all, in fact.

When you decide to use the username/password authentication, you will have to send it with every request; when you use a token, you will need to send username/password once and you will get a secret string of bytes which can be reused until it expires on its own or when the OpenStack cloud provider's administrator revokes it. With the API key you do not have to send your username/password credentials, but you are responsible for expiring the key.

Remembering Basic Security Hygiene

Please make sure that the scripts you are pasting your credentials into are not readable by other users because the authentication tokens, the API keys, as well as your username and password are literally the key to your account and with it anyone who gets a hold of it will be able to wreak havoc with your cloud.

If you suspect that the worst might have happened, revoke the compromised credentials and generate new ones by folowing the instructions provided by your OpenStack cloud provider. It will make all scripts, rouge and legitimate, inoperable, so once you cut them off, you'll need to fix the security holes and then replace the compromised credentials in your scripts with the new ones.

Getting the Authentication Token

In order to obtain an API token you will need to request it from your OpenStack cloud provider's authentication server. Your provider will have to give you the URL of their authentication server as well as your username and password.

The OpenStack Compute API is implemented using a convention known as "representational state transfer" (REST), which is a way of using various HTTP methods (GET, POST, PUT, DELETE) to manage resources described using paths to those resources. It sounds more complex than it is in reality.

The following example shows how to use the curl command to obtain the authentication token using the HTTP GET method:

```
$ curl -d '{"passwordCredentials": {"username": "joe", "password": "shhh"}}' -
H "Content-type: application/json" http://localhost:5000/v2.0/tokens
```

- joe — your username, replace joe without your username, unless it is joe.
- shhh — the password for your OpenStack account, please do not use shhh.
- localhost:5000 — the URL of the authentication server, localhost is just a placeholder, please use an IP address or a full hostname (e.g. somehost.example.com). Please pay attention to the port number, it does matter. If your provider fails to tell you which port should be used, try 5000 and 5001 as these are typically used by the OpenStack authentication servers to listen for authentication requests.
- /v2.0/tokens — the path to the token provider.

What you will get in return should look like the following stream of code:

```
{"auth": {"token": {"expires": "2015-02-05T00:00:00", "id": "999888777666"},
 "serviceCatalog": {"glance": [{"adminURL": "http://10.0.2.15:9292/
v1", "region": "RegionOne", "internalURL": "http://10.0.2.15:9292/v1",
 "publicURL": "http://10.0.2.15:9292/v1"}], "identity": [{"adminURL": "http:/
/10.0.2.15:5001/v2.0", "region": "RegionOne", "internalURL": "http://10.
0.2.15:5000/v2.0", "publicURL": "http://10.0.2.15:5000/v2.0"}], "nova":
 [{"adminURL": "http://10.0.2.15:8774/v1.1/openstack", "region": "RegionOne",
 "internalURL": "http://10.0.2.15:8774/v1.1/openstack", "publicURL": "http://
10.0.2.15:8774/v1.1/openstack"}]}}}
```

It is hard to read, so here's the same output formatted for your convenience:

```
{
  "auth": {
    "token": {
      "expires": "2015-02-05T00:00:00",
      "id": "999888777666"
    },
    "serviceCatalog": {
      "glance": [
        {
          "adminURL": "http://10.0.2.15:9292/v1",
```

```
          "region": "RegionOne",
          "internalURL": "http://10.0.2.15:9292/v1",
          "publicURL": "http://10.0.2.15:9292/v1"
        }
      ],
      "nova": [
        {
          "adminURL": "http://10.0.2.15:8774/v1.1/openstack",
          "region": "RegionOne",
          "internalURL": "http://10.0.2.15:8774/v1.1/openstack",
          "publicURL": "http://10.0.2.15:8774/v1.1/openstack"
        }
      ],
      "identity": [
        {
          "adminURL": "http://10.0.2.15:5001/v2.0",
          "region": "RegionOne",
          "internalURL": "http://10.0.2.15:5000/v2.0",
          "publicURL": "http://10.0.2.15:5000/v2.0"
        }
      ]
    }
  }
}
```

The reponse contains the token itself (the id key in the token dictionary), the token expiry date/time and the catalog of services you can talk to that accept the token. The catalog lists the names of the services you can access together with their region names, the administration URL, the internal URL and the public URL for each region. Regions are groups of servers, server farms, etc. usually located in the same geogrphical area.

Each region is represented as a separate dictionary and those dictionaries are items in a list that groups region information for each service. This is an important exception to the overall composition of the JSON response which is arranged in a form of a nested key/value directory.

Although this is not going to be a frequent occurence in the real world, you may get bitten by server misconfiguration in test environments. If your code extracts the endpoint URLs automatically, and later hangs or starts reporting strange network errors, check if the URLs and port numbers returned by the authentication server match the reality. It is easy to forget to set the right hostname/port. The OpenStack configuration is a fairly complex process and it is easy to get confused. If you get an OpenStack training environment in the form of a VirtualBox or VMware virtual machine, try replacing whatever IP addresses or hostnames are returned by the authentication server with localhost or run ifconfig to see which IP addresses have been assigned to the virtual machine.

What About HTTPS?

All calls to the OpenStack Compute API presented in the examples included in this book use unencrypted HTTP connections for the convenience of testing things. In real life you will most likely be using encrypted HTTPS connections, which need some additional care.

When you want to call your OpenStack provider's API servers you will most likely need to use a secure HTTPS connection. You will recognize it by the https prefix. If you are given an http URL, try replacing it with https and see if that one works.

If you run into problems with SSL certificates while using curl, read the instructions published at the following URL:

- http://curl.haxx.se/docs/sslcerts.html

Most problems with SSL are caused by SSL certificate misconfiguration on the server side or by the use of self-generated SSL certificates.

Using HTTPS in Python scripts is explained later in this chapter.

Using Python to Obtain the Authentication Token

When you want to obtain an API token you can do it using two standard Python modules: httplib and json. Both are available in all major branches of Python: 2.6.x, 2.7.x, and 3.x.x. If you don't like them, you can always use requests and simplejson.

```
#!/usr/bin/python

import httplib
import json

# arguments

## make sure that url is set to the actual hostname/IP address,
## port number

url = "192.168.10.1:5000"

## make sure that osuser is set to your actual username, "admin"
## works for test installs on virtual machines, but it's a hack

osuser = "joe"

## use something else than "shhh" for you password

ospassword = "shhh"

params = '{"passwordCredentials":{"username":osuser, "password":ospassword}}'

headers = {"Content-Type": "application/json"}

# HTTP connection

conn = httplib.HTTPConnection(url)
conn.request("POST", "/v2.0/tokens", params, headers)

# HTTP response

response = conn.getresponse()
data = response.read()
dd = json.loads(data)

conn.close()

apitoken = dd['auth']['token']['id']

print "Your token is: %s" % apitoken
```

When you run gettoken.py you should see the following output (the token will hopefully be different):

```
$ python ./gettoken.py
Your token is: 999888777666
```

Getting the API Endpoint URL

The API token is only one essential piece of information you need to know to use the OpenStack Compute API. Another is the URL of the Nova API end point. That's where you will send your calls to the OpenStack Compute API accompanied by the token you just obtained. That information may be given to you by your OpenStack cloud provider or you may be able to extract that information from the authentication server's response:

```
#!/usr/bin/python

import httplib
import json

# arguments

## change to False when you are using the test environment

usehttps = True

## make sure that url is set to the actual hostname/IP address,
## port number

url = "192.168.10.1:5000"

## make sure that osuser is set to your actual username, "admin"
## works for test installs on virtual machines, but it's a hack

osuser = "joe"

## use something else than "shhh" for you password

ospassword = "shhh"

params = '{"passwordCredentials":{"username":osuser, "password":ospassword}}'

headers = {"Content-Type": "application/json"}

# HTTP connection

if (usehttps == True):
    conn = httplib.HTTPSConnection(url, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn = httplib.HTTPConnection(url)

conn.request("POST", "/v2.0/tokens", params, headers)

# HTTP response

response = conn.getresponse()
```

```
data = response.read()
dd = json.loads(data)

conn.close()

apitoken = dd['auth']['token']['id']
apiurl = dd['auth']['serviceCatalog']['nova'][0]['publicURL']

print "Your token is: %s" % apitoken
print "Your Nova URL is: %s" % apiurl
```

When you run that script, you should see the following output. The token will most likely be something different than 999888777666 and the Nova URL will not always contain 10.0.2.15:

```
$ ./gettokenurl.py
Your token is: 999888777666
Your Nova URL is: http://10.0.2.15:8774/v1.1/openstack
```

Getting Your Nova Credentials -- The Drop-In Script Snippet for Your Python Scripts

You will be coming back to the token and API URL extraction code over and over again, which is why I decided to place it in a separate section of this book. That way I can make the examples shorter and you can focus of the important stuff. Here's the code in all it's glory:

```
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

### Get OpenStack Credentials

# arguments

## change to False when you are using the test environment

usehttps = True

## make sure that url1 is set to the actual hostname/IP address,
## port number

url1 = "localhost:5000"

## make sure that osuser is set to your actual username, "admin"
## works for test installs on virtual machines, but it's a hack

osuser = "joe"

## use something else than "shhh" for you password

ospassword = "shhh"

params1 = '{"passwordCredentials":{"username":osuser, "password":ospassword}}'
headers1 = {"Content-Type": "application/json"}

# HTTP connection #1

if (usehttps == True):
```

```
    # set key_file and cert_file to wherever the key and cert files
    # are located
    conn1 = httplib.HTTPSConnection(url1, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn1 = httplib.HTTPConnection(url1)

conn1.request("POST", "/v2.0/tokens", params1, headers1)

# HTTP response #1

response1 = conn1.getresponse()
data1 = response1.read()
dd1 = json.loads(data1)

conn1.close()

# extract token and url

apitoken = dd1['auth']['token']['id']
apiurl = dd1['auth']['serviceCatalog']['nova'][0]['publicURL']
apiurlt = urlparse(dd1['auth']['serviceCatalog']['nova'][0]['publicURL'])

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
```

第 3 章 Servers

Servers are the cornerstone of the OpenStack Compute cloud infrastructure. While the future implementation of OpenStack will most likely offer independent file storage, content delivery networks, or cloud networking components, servers are what we have right now.

Creating Servers

When you create an account with your OpenStack Compute provider you will need to create your servers yourself, either by booting them via a web interface or with an API call. The call to create a server must use the POST method and the body of the request must be a JSON bundle that contains three essential arguments:

- name — the name of the new server.
- imageRef — the URL of the server image.
- flavorRef — the URL of the server flavor.

A short explanation is in order. Flavors contain definitions of the virtual hardware that will be used to run the server system installed from an image. Images can be clean operating systems, or they can be modified by yourself in order to use them as virtual cookie cutters. For example, you could use Ubuntu to configure a web server and save the contents of the filesystem as an image that will be used to create as many additional web servers as necessary. This is faster, easier and more reliable than booting a clean system and configuring it from scratch every time you want to add a new web server.

Images are not tied to any particular virtual hardware flavor. The only limit is the size of the storage and the amount of RAM available for a particular flavor.

Whenever one of those three components are missing, the OpenStack Compute API will complain with the following messages:

- missing name

You forgot to name the new server:

```
{
  "badRequest": {
    "message": "Server name is an empty string",
    "code": 400
  }
}
```

- missing imageRef

You forgot the URL of the server image or used the wrong one:

```
{
  "badRequest": {
```

```
    "message": "Cannot find requested image : Image 0 could not be found.",
    "code": 400
  }
}
```

・missing flavorRef

You forgot the URL of the virtual machine flavor or used the wrong one:

```
{
  "badRequest": {
    "message": "Invalid flavorRef provided.",
    "code": 400
  }
}
```

The value of name is easy to figure out, it is what you want to call your server. Typically that would be some sort of a shortcut for a function that the server performs, e.g. when I create web servers using the Tornado Web server, I usually call the hosts tornado001, tornado002, etc. or I might call them web000, www000, whatever.

But imageRef and flavorRef cannot be just any URL. You need to ask the OpenStack Compute API for the list of images and flavors. That list will contain the URLs accompanied by some additional information that you can use to automate things like switching to a flavor that offers more storage or higher CPU performance.

Listing Images

How do you know what images are available for building new servers?

You can list available server images by sending your request to the /images URL using the GET method. The following example shows how it is done using the curl command:

```
$ curl -v -H "X-Auth-Token:999888777666" http://localhost:8774/v1.1/openstack/
images
```

As you can see, you must include the X-Auth-Token header with a valid token otherwise the OpenStack Compute API will return the 401 Unauthorized error.

When all goes well, you will get a lump of JSON that looks a bit like this:

```
{"images": [{"id": 3, "links": [{"href": "http://localhost:8774/v1.1/
openstack/images/3", "rel": "self"}, {"href": "http://localhost:8774/
openstack/images/3", "rel": "bookmark"}], "name": "ubuntu-11.04-server"},
 {"id": 2, "links": [{"href": "http://localhost:8774/v1.1/openstack/images/2",
 "rel": "self"}, {"href": "http://localhost:8774/openstack/images/2", "rel":
 "bookmark"}], "name": "ubuntu-11.04-initrd"}, {"id": 1, "links": [{"href":
 "http://localhost:8774/v1.1/openstack/images/1", "rel": "self"}, {"href":
 "http://localhost:8774/openstack/images/1", "rel": "bookmark"}], "name":
 "ubuntu-11.04-kernel"}]}
```

Here is a version that is easier to read:

```
```

```
{
  "images": [
    {
      "id": 3,
      "links": [
        {
          "href": "http://localhost:8774/v1.1/openstack/images/3",
          "rel": "self"
        },
        {
          "href": "http://localhost:8774/openstack/images/3",
          "rel": "bookmark"
        }
      ],
      "name": "ubuntu-11.04-server"
    },
    {
      "id": 2,
      "links": [
        {
          "href": "http://localhost:8774/v1.1/openstack/images/2",
          "rel": "self"
        },
        {
          "href": "http://localhost:8774/openstack/images/2",
          "rel": "bookmark"
        }
      ],
      "name": "ubuntu-11.04-initrd"
    },
    {
      "id": 1,
      "links": [
        {
          "href": "http://localhost:8774/v1.1/openstack/images/1",
          "rel": "self"
        },
        {
          "href": "http://localhost:8774/openstack/images/1",
          "rel": "bookmark"
        }
      ],
      "name": "ubuntu-11.04-kernel"
    }
  ]
}
```


Each item on the images list represents a single server image. It is a dictionary with three keys:

- id — the numeric ID of the server image.
- links — the list of the URLs of the locations where you can find the image.
- name — a descriptive name of the image.

Here's how one server image is represented in that response:


```
{
```

```
  "id": 1,
  "links": [
    {
      "href": "http://localhost:8774/v1.1/openstack/images/1",
      "rel": "self"
    },
    {
      "href": "http://localhost:8774/openstack/images/1",
      "rel": "bookmark"
    }
  ],
  "name": "ubuntu-11.04-kernel"
}
```

Please note that this is not a complete set of information about server images that you can extract using the OpenStack Compute API, but it is enough to create a server unless you are trying to use a freshly created image, which hasn't been activated yet. For more information about handling such cases and how to work with server images in general, read the Images chapter later in this book.

One question that programmers sometimes ask is which kind of server image identification is more reliable to rely upon, the server image id or its name? Personally, I rely on the names of server images, assuming that should some administrative mishap happen and the server image store got reorganized, names are less likely to be altered than the numeric IDs.

Another argument for using names to identify server images is handling errors and ensuring operational continuity. If you want to make sure that the region or the provider you want to boot your server with has the right server image, you will want to use a unique name instead of a numeric ID, which can differ between OpenStack regions and providers. As always, check and double-check these things in your code. The best strategy is to use automation to create and save same server images in each region and with each OpenStack provider you plan to use. This is necessary, because you cannot transfer server images between OpenStack regions or providers. The automation tool of choice for Python programmers is Fabric, which you can download from the following site:

- http://fabfile.org

Listing Flavors

Once you have the server image URL, you will need the URL of the virtual machine flavor. You can list available flavors by sending your request to the /flavors URL using the GET method. The following example shows how it is done using the curl command:

```
$ curl -v -H "X-Auth-Token:999888777666" http://localhost:8774/v1.1/openstack/
flavors
```

As with every call you make to the OpenStack Compute API you must include the X-Auth-Token header with a valid token otherwise the OpenStack Compute API will return the 401 Unauthorized error.

When all goes well, you will get a JSON string that looks like the one below:

```
```

```
{"flavors": [{"id": 3, "links": [{"href": "http://localhost:8774/v1.1/
openstack/flavors/3", "rel": "self"}, {"href": "http://localhost:8774/
openstack/flavors/3", "rel": "bookmark"}], "name": "m1.medium"}, {"id":
 4, "links": [{"href": "http://localhost:8774/v1.1/openstack/flavors/4",
 "rel": "self"}, {"href": "http://localhost:8774/openstack/flavors/4", "rel":
 "bookmark"}], "name": "m1.large"}, {"id": 1, "links": [{"href": "http://
localhost:8774/v1.1/openstack/flavors/1", "rel": "self"}, {"href": "http://
localhost:8774/openstack/flavors/1", "rel": "bookmark"}], "name": "m1.tiny"},
 {"id": 5, "links": [{"href": "http://localhost:8774/v1.1/openstack/flavors/
5", "rel": "self"}, {"href": "http://localhost:8774/openstack/flavors/5",
 "rel": "bookmark"}], "name": "m1.xlarge"}, {"id": 2, "links": [{"href":
 "http://localhost:8774/v1.1/openstack/flavors/2", "rel": "self"}, {"href":
 "http://localhost:8774/openstack/flavors/2", "rel": "bookmark"}], "name":
 "m1.small"}]}
```

Here is a version that is easier to read:

```
{
  "flavors": [
    {
      "id": 3,
      "links": [
        {
          "href": "http://localhost:8774/v1.1/openstack/flavors/3",
          "rel": "self"
        },
        {
          "href": "http://localhost:8774/openstack/flavors/3",
          "rel": "bookmark"
        }
      ],
      "name": "m1.medium"
    },
    {
      "id": 4,
      "links": [
        {
          "href": "http://localhost:8774/v1.1/openstack/flavors/4",
          "rel": "self"
        },
        {
          "href": "http://localhost:8774/openstack/flavors/4",
          "rel": "bookmark"
        }
      ],
      "name": "m1.large"
    },
    {
      "id": 1,
      "links": [
        {
          "href": "http://localhost:8774/v1.1/openstack/flavors/1",
          "rel": "self"
        },
        {
          "href": "http://localhost:8774/openstack/flavors/1",
          "rel": "bookmark"
        }
      ],
```

```
      "name": "m1.tiny"
    },
    {
      "id": 5,
      "links": [
        {
          "href": "http://localhost:8774/v1.1/openstack/flavors/5",
          "rel": "self"
        },
        {
          "href": "http://localhost:8774/openstack/flavors/5",
          "rel": "bookmark"
        }
      ],
      "name": "m1.xlarge"
    },
    {
      "id": 2,
      "links": [
        {
          "href": "http://localhost:8774/v1.1/openstack/flavors/2",
          "rel": "self"
        },
        {
          "href": "http://localhost:8774/openstack/flavors/2",
          "rel": "bookmark"
        }
      ],
      "name": "m1.small"
    }
  ]
}
```

The list of virtual machine flavors is arranged in a way similar to the list of server images. Each item on the flavors list represents a single virtual machine flavor. It is a dictionary with three keys:

- id — the numeric ID of the flavor.
- links — the list of the URLs of the locations where you can find the flavor.
- name — a descriptve name of the flavor.

Here's how one virtual machine flavor is represented in that response:

```
{
  "id": 1,
  "links": [
    {
      "href": "http://localhost:8774/v1.1/openstack/flavors/1",
      "rel": "self"
    },
    {
      "href": "http://localhost:8774/openstack/flavors/1",
      "rel": "bookmark"
    }
  ],
  "name": "m1.tiny"
},
```

Booting a New Server

You already know how to obtain the authentication token, the URL of the OpenStack Compute API, and the URLs of both the server image and the flavor. You already know all you need to know to make the basic POST request to create a server. We will now create now a Python script that does that job for us. First, we need to obtain the authentication token and the URL of the OpenStack Compute API (aka. nova):

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
```

With the token and the URL we can now ask the OpenStack Compute API server for the image reference:

```
###
### Get server image reference
###

# HTTP connection #2

url2 = apiurlt[1]
params2 = urllib.urlencode({})
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("GET", "%s/images" % apiurlt[2], params2, headers2)

# HTTP response

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()

###
### Server parameters
###

# Server name
```

```
sname = "tornado001"

# Server image URL

n = len(dd2["images"])
m = range(n)

for i in m:
    if dd2["images"][i]["id"] == 1:
        sImageRef = dd2["images"][i]["links"][0]["href"]
```

All we need now is the flavor URL:

```
# Flavor URL

# HTTP connection #3

params3 = urllib.urlencode({})
headers3 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn3 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn3 = httplib.HTTPConnection(url2)

conn3.request("GET", "%s/flavors" % apiurlt[2], params3, headers3)

# HTTP response #3

response3 = conn3.getresponse()
data3 = response3.read()
dd3 = json.loads(data3)

conn3.close()

n = len(dd3["flavors"])
m = range(n)

for i in m:
    if dd3["flavors"][i]["id"] == 1:
        sFlavorRef = dd3["flavors"][i]["links"][0]["href"]
```

We can leave other parameters empty for now:

```
###
### server metadata
###

sMetadata = {}

###
### server personalization
###

sPersonalityPath = ""
sPersonalityContents = ""
```

```
sPersonality = [ { "path":sPersonalityPath, "contents":base64.
b64encode( sPersonalityContents ) } ]
```

All that's left to do is a call to the OpenStack Compute API to create a new server:

```
s = { "server": { "name": sname, "imageRef": sImageRef, "flavorRef":
 sFlavorRef, "metadata": sMetadata, "personality": sPersonality } }

sj = json.dumps(s)

# HTTP connection #4

params4 = sj
headers4 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

conn4 = httplib.HTTPConnection("localhost:8774")
conn4.request("POST", "%s/servers" % apiurlt[2], params4, headers4)

# HTTP response #4

response4 = conn4.getresponse()
data4 = response4.read()
dd4 = json.loads(data4)

conn4.close()

print json.dumps(dd4, indent=2)
```

When you run that script, you will get back a JSON string that looks simiar to the one below (shown expanded for better readability):

```
{
  "server": {
    "status": "BUILD",
    "updated": "2011-12-04T21:18:55Z",
    "hostId": "",
    "user_id": "admin",
    "name": "tornado001",
    "links": [
      {
        "href": "http://localhost:8774/v1.1/openstack/servers/1",
        "rel": "self"
      },
      {
        "href": "http://localhost:8774/openstack/servers/1",
        "rel": "bookmark"
      }
    ],
    "created": "2011-12-04T21:18:55Z",
    "tenant_id": "openstack",
    "image": {
      "id": "1",
      "links": [
        {
          "href": "http://localhost:8774/openstack/images/1",
          "rel": "bookmark"
        }
      ]
    },
```

```
    "adminPass": "MrhcNkNqAeVnnUE7",
    "uuid": "8fec45a8-92fb-4840-8f2a-c64e1be49dee",
    "accessIPv4": "",
    "metadata": {
      "Server Name": "Tornado"
    },
    "accessIPv6": "",
    "key_name": null,
    "progress": 0,
    "flavor": {
      "id": "1",
      "links": [
        {
          "href": "http://localhost:8774/openstack/flavors/1",
          "rel": "bookmark"
        }
      ]
    },
    "config_drive": "",
    "id": 1,
    "addresses": {}
  }
}
```

Congratulations! You have a new server.

When you create a new server you are going to get a lot of interesting information about it in return:

- status: set initially to BUILD, later set to ACTIVE when the server is ready to use.
- updated: set initially to the server creation date, will be updated every time you modify the server.
- hostId: undocumented.
- user_id: the username of the user who created the server.
- name: a short server name assigned by whoever created the server (you), in this case it is tornado001.
- links: the URLs pointing to the server.
- created: set to the server creation date.
- tenant_id: the name of the tenant that the server lives with, in this case it's openstack.
- image: basics data information about the operating system image used to build the server.
- adminPass: the initial root/Administrator password, set automatically for you. It is possible to change it afterwards.
- uuid: the Universally Unique Identifier (UUID) of the server.
- accessIPv4: the public IPv4 address of the server. You will use it to access your server.
- metadata: a dictionary that stores server metadata. This can be edited by yourself as explained later in this book.

- accessIPv6: the public IPv6 address of the server.
- key_name: undocumented.
- progress: an integer value of 0 through 100. It starts with 0 and increases as the server is getting created. It is usually enough to monitor the status property of a new server, but if it takes longer than a couple of miutes to build, the progress indicator can provide additional information that may help debug the problem.
- flavor: basic data information about the virtual hardware flavor used to build the server.
- config_drive: undocumented.
- id — the id of the server within your own OpenStack account realm. The numbering starts with 1.

Using Curl to Boot a New Server

If you want to use curl to create a new server, this is how you'd structure your request:

```
$ curl -v -X POST -H "X-Auth-Token:999888777666" -H "Content-type:application/
json" -d '{"server": {"flavorRef": "http://localhost:8774/v1.1/openstack/
flavors/1", "personality": [{"path": "", "contents": ""}], "name":
 "tornado001", "imageRef": "http://localhost:8774/v1.1/openstack/images/
1", "metadata": {"Server Name": "Tornado"}}}' http://localhost:8774/v1.1/
openstack/servers
```

The result on that request will be a new server, whose description will be similar to the one discussed in the previous section.

Customizing Your Server: Adding Metadata

Short server names like tornado000 may be good enough to describe a small set of servers, but they become insufficient when you need to manage dozens of machines. OpenStack Compute API allows us to add a number of metadata entries to each server at the time of its creation.

If you go back to the section with the example of using Python to create a new server, you will see the following snippet of code:

```
...

###
### server metadata
###

sMetadata = {}

###
### server personalization
###

sPersonalityPath = ""
sPersonalityContents = ""
```

```
sPersonality = [ { "path":sPersonalityPath, "contents":base64.
b64encode( sPersonalityContents ) } ]

s = { "server": { "name": sname, "imageRef": sImageRef, "flavorRef":
 sFlavorRef, "metadata": sMetadata, "personality": sPersonality } }

sj = json.dumps(s)

...
```

Because providing server metadata is optional, the example sets it to an empty dictionary. When you want to add metadata at server creation time, you will need to provide a dictionary of upto 5 key/value pairs. For example, if you wanted to add more information about the Tornado web server you plan to create you could set sMetadata to something like this:

```
...

###
### server metadata
###

sMetadata = {"function":"dynamic content", "master":"no", "belongs-to":"load
 balanced group no 1", "admin":"Joe Speedoo", "backup":"no"}

###
### server personalization
###

sPersonalityPath = ""
sPersonalityContents = ""
sPersonality = [ { "path":sPersonalityPath, "contents":base64.
b64encode( sPersonalityContents ) } ]

s = { "server": { "name": sname, "imageRef": sImageRef, "flavorRef":
 sFlavorRef, "metadata": sMetadata, "personality": sPersonality } }

sj = json.dumps(s)

...
```

The keys and values you choose can be anything you like, but it is a good idea to develop a schema that can be reused. For example, in the code fragment shown above, I use function to store a short description of what the server does; master to indicate if the server is a copy of the master server or the master server itself; belongs-to to indicate, which group of load-balanced servers the new machine will belog to; admin to store the name of the server administrator; and, backup to indicate if this server needs to be backed up.

The amount of metadata information assigned to each OpenStack server is currently limited to: 5 keys each holding values no larger than 10240 bytes.

Customizing Your Server: Adding Custom Files

There are three ways of customizing newly created servers:

- log into a server and customize it by hand (not very efficient).

- create a Fabric script and customize it automatically.
- preload server configuration scripts and (optionally) reboot it, wait, test if everything went fine.

While you have to take care of the first two solutions, OpenStack Compute API can help with the last one. You can write up to 5 files, each no larger than 10KB (10240 bytes) to the freshly installed filesystem. They can be located anywhere, e.g. you could do something trivial like change the contents of /etc/motd, or you could add a system configuration script to /etc/init.d just remember to include logic that does the installation once and not at every reboot, unless that's what you want.

However you want to use server personalization, you need to put the personalization files into a dictionary whose keys are paths to the files located on the server. Each key has a value, which in this case is the contents of the destination file, which needs to be encoded as Base64.

To learn how it works in practice, we need to go back to the example showing how to create a new server and focus our attention of the following snippet of code:

```
...

###
### server metadata
###

sMetadata = {"function":"dynamic content", "master":"no", "belongs-to":"load
 balanced group no 1", "admin":"Joe Speedoo", "backup":"no"}

###
### server personalization
###

sPersonalityPath = ""
sPersonalityContents = ""
sPersonality = [ { "path":sPersonalityPath, "contents":base64.
b64encode( sPersonalityContents ) } ]

s = { "server": { "name": sname, "imageRef": sImageRef, "flavorRef":
 sFlavorRef, "metadata": sMetadata, "personality": sPersonality } }

sj = json.dumps(s)

...
```

If you carefully examine the value of sPersonality, you will notice that it is a list of dictionaries with two keys (path and contents) each. You can define upto 5 such dictionaries. Here's how you could do it if you wanted to copy to a newly created server 5 local files (their names and locations do not have to match the names and the locations of their destination):

```
...

###
### server metadata
###
```

```
sMetadata = {"function":"dynamic content", "master":"no", "belongs-to":"load
 balanced group no 1", "admin":"Joe Speedoo", "backup":"no"}

###
### server personalization
###

sPPath1 = "/home/joe/.profile"
sPContents1F = open('joe.profile')
sPContents1 = sPContents1F.read()
sPCont1 = base64.b64encode( sPContents1 )

sPPath2 = "/etc/motd"
sPContents2F = open('joe.motd')
sPContents2 = sPContents2F.read()
sPCont2 = base64.b64encode( sPContents2 )

sPPath3 = "/etc/nginx/nginx.conf"
sPContents3F = open('joe.nginx.conf')
sPContents3 = sPContents3F.read()
sPCont3 = base64.b64encode( sPContents3 )

sPPath4 = "/etc/ssh/sshd_config"
sPContents4F = open('joe.sshd_config')
sPContents4 = sPContents4F.read()
sPCont4 = base64.b64encode( sPContents4 )

sPPath5 = "/etc/sysctl.conf"
sPContents5F = open('joe.sysctl.conf')
sPContents5 = sPContents5F.read()
sPCont5 = base64.b64encode( sPContents5 )

sPersonality = [ { "path":aPPath1, "contents":sPCont1 }, { "path":aPPath2,
 "contents":sPCont2 }, { "path":aPPath3, "contents":sPCont3 },
 { "path":aPPath4, "contents":sPCont4 }, { "path":aPPath5,
 "contents":sPCont5 } ]

s = { "server": { "name": sname, "imageRef": sImageRef, "flavorRef":
 sFlavorRef, "metadata": sMetadata, "personality": sPersonality } }

sj = json.dumps(s)

...
```

What if five files are not enough? You can add more after the server boots. You can do it the hard way using scp or you can write a Fabric script. Use Fabric. It lives here:

- http://fabfile.org

Server Limits

Although cloud-based servers give us a lot of flexibility, they still live in data centers and need to be managed in order to avoid hoarding of the data center resources. This is not a flaw, but the fact of life, even private server clouds have their limits.

Every OpenStack cloud provider is free to set their own limits, but the ones suggested in the official documentation are:

- maxTotalRAMSize — total amount of RAM of all active servers: 51200MB (50GB), this could be up to two hundred 256MB instances, or any mixture of other servers.
- maxServerMeta — maximum number of metadata entries per server: 5.
- maxImageMeta — maximum number of metadata entries per image: 5.
- maxPersonality — maximum number of personalisation files per server: 5.
- maxPersonalitySize — maximum size of each personalization file: 10240 bytes (10KB).

Modifying Live Servers

Once a server is created, it is by no means set in stone. Apart from the obvious modifications to the operating system you can also modify the virtual hardware, the server's metadata, as well as perform some basic maintenance tasks.

How you modify the operating system is up to you and up to the ways of the operating system, the rest can be done via the OpenStack Compute API.

Listing Servers

Before you can modify anything, you should have a good look at it.

When you want to check how many servers you have active, you need to use the GET mthod when you call the /server URL. When you are testing things, use the curl command to get the raw JSON reply:

```
$ curl -v -H "X-Auth-Token:999888777666" http://localhost:8774/v1.1/openstack/
servers
```

When you have no active servers, the result will be a dictionary with one key—servers—whose value is an empty list:

```
{
  "servers": []
}
```

When there are more servers active, you will see a longer list of all servers that you can manage:

```
{
  "servers": [
    {
      "id": 2,
      "uuid": "fcdeba5b-40f7-4f51-9404-8a41951739b2",
      "links": [
        {
          "href": "http://localhost:8774/v1.1/openstack/servers/2",
          "rel": "self"
        },
        {
          "href": "http://localhost:8774/openstack/servers/2",
          "rel": "bookmark"
```

```
        }
      ],
      "name": "tornado002"
    },
    {
      "id": 1,
      "uuid": "8fec45a8-92fb-4840-8f2a-c64e1be49dee",
      "links": [
        {
          "href": "http://localhost:8774/v1.1/openstack/servers/1",
          "rel": "self"
        },
        {
          "href": "http://localhost:8774/openstack/servers/1",
          "rel": "bookmark"
        }
      ],
      "name": "tornado001"
    }
  ]
}
```

For a better view, here is a single server entry in the servers list:

```
{
  "id": 1,
  "uuid": "8fec45a8-92fb-4840-8f2a-c64e1be49dee",
  "links": [
    {
      "href": "http://localhost:8774/v1.1/openstack/servers/1",
      "rel": "self"
    },
    {
      "href": "http://localhost:8774/openstack/servers/1",
      "rel": "bookmark"
    }
  ],
  "name": "tornado001"
}
```

The keys in a dictionary that describes each server have the following meaning:

- id — the numeric ID of the server.
- uuid — the universally unique identifier of the server.
- links — the list of URLs that point to the server.
- name — a descriptive name of the server.

It is worth poinitng out a few things: the server id is unique within your account with the OpenStack cloud provider. Other users with the same provider may (and will) have the same server id numbers. They are assigned in the order of server creation. Similarly, the the value of the name key does not have to be unique, even within your own account.

The only unique identifier of your server will be its uuid. It is assigned automatically by you OpenStack cloud provider's system using an algorithm that is supposed to guarantee that the IDs it creates do not clash. You can read more about UUIDs on Wikipedia:

- http://en.wikipedia.org/wiki/Universally_unique_identifier

Listing Servers Using Python

When you want to use Python to list server information, the procedure is a bit more complex, but you do get access to the items returned by the OpenStack Compute API.

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Get the list of servers
###

# HTTP connection #2

url2 = apiurlt[1]
params2 = urllib.urlencode({})
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("GET", "%s/servers" % apiurlt[2], params2, headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()

print dd2
```

When you want to access specific keys in the dictionary of servers, you will need to either hard-code the paths to specific pieces of information, or use some simple heuristics, e.g. in the following snippet of code I assume that you already have the list of server information:

```
# abort when there are no servers
```

```
if dd2['servers'] == []:
    print 'Error: No servers found'
    exit(1)

# iterate through server entries until you find the server by name and
# then get its ID and URL

dd2r = range(len(dd2['server']))

for n in dd2r:
    if dd2['servers'][n]['name'] == 'tornado001':
        print "Server ID",
        print str(dd2['servers'][n]['id'])
        print "Server URL",
        print dd2['servers'][n]['links'][0]['href']
```

Getting Detailed Server Information

A call to /servers returns the list of all servers and can be very long, which is why the JSON response contain only that data which can be used in other calls to update, rebuild, or delete a server.

You can learn much more about a particular server by calling a different URL. It's /server/id where id is the value of the id key in the server description. Try it using curl:

```
$ curl -v -H "X-Auth-Token:999888777666" http://localhost:8774/v1.1/openstack/servers/1
```

What you will get in return is a JSON response with a lot more information about server whose id is set to 1:

```
{
  "server": {
    "status": "ACTIVE",
    "updated": "2011-12-04T21:19:11Z",
    "hostId": "ea05d3b6c34b9baae7ad5ce482814e040305b6c8fb67034a46908720",
    "user_id": "admin",
    "name": "tornado001",
    "links": [
      {
        "href": "http://localhost:8774/v1.1/openstack/servers/1",
        "rel": "self"
      },
      {
        "href": "http://localhost:8774/openstack/servers/1",
        "rel": "bookmark"
      }
    ],
    "created": "2011-12-04T21:18:55Z",
    "tenant_id": "openstack",
    "image": {
      "id": "1",
      "links": [
        {
          "href": "http://localhost:8774/openstack/images/1",
          "rel": "bookmark"
```

```
        }
      ]
    },
    "uuid": "8fec45a8-92fb-4840-8f2a-c64e1be49dee",
    "accessIPv4": "",
    "metadata": {
      "Server Name": "Tornado"
    },
    "accessIPv6": "",
    "key_name": null,
    "progress": 100,
    "flavor": {
      "id": "1",
      "links": [
        {
          "href": "http://localhost:8774/openstack/flavors/1",
          "rel": "bookmark"
        }
      ]
    },
    "config_drive": "",
    "id": 1,
    "addresses": {
      "public": [
        {
          "version": 4,
          "addr": "192.168.100.2"
        }
      ],
      "private": [
        {
          "version": 4,
          "addr": "192.168.200.2"
        }
      ]
    }
  }
}
```


Here is a guide to all those tasty bits of information:

- status: set initially to BUILD, later set to ACTIVE when the server is ready to use.
- updated: set initially to the server creation date, will be updated every time you modify the server.
- hostId: undocumented.
- user_id: the username of the user who created the server.
- name: a short server name assigned by whoever created the server (you), in this case it is tornado001.
- links: the URLs pointing to the server.
- created: set to the server creation date.
- tenant_id: the name of the tenant that the server lives with, in this case it's openstack.

- image: basics data information about the operating system image used to build the server.
- uuid: the Universally Unique Identifier (UUID) of the server.
- accessIPv4: the public IPv4 address of the server. You will use it to access your server.
- metadata: a dictionary that stores server metadata. This can be edited by yourself as explained later in this book.
- accessIPv6: the public IPv6 address of the server.
- key_name: undocumented.
- progress: an integer value of 0 through 100. It starts with 0 and increases as the server is getting created. It is usually enough to monitor the status property of a new server, but if it takes longer than a couple of miutes to build, the progress indicator can provide additional information that may help debug the problem.
- flavor: basic data information about the virtual hardware flavor used to build the server.
- config_drive: undocumented.
- id — the id of the server within your own OpenStack account realm. The numbering starts with 1.

The list show above is almost an exact copy of the list shown in the chapter on server creation. One missing field is adminPass, which is never returned after the server starts. This is done for security reasons. If you want to change the server root/Administrator password, you need to do it via the web interface or using a special OpenStack Compute API call.

Getting Detailed Server Information Using Python

What if you want to get all details of a particular server using Python? Here's how you do it:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Get the full description of a server
```

```
###

# HTTP connection #2

servID = 1      # server ID
url2 = apiurlt[1]
params2 = urllib.urlencode({})
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("GET", "%s/servers/%s" % (apiurlt[2], servID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()

print dd2
```

Listing Server Addresses

One piece of information that is looked up more often than others is server IP address. Or addresses, as is the case with all OpenStack Compute servers. Each machine has at least two IPv4 addresses: one private and one public (aka. routable). It is also reasonable to assume that each server will have its own private and public IPv6 address.

You can list them using a GET request for the /servers/id/ips URL. For example, if you were to use curl to list the IP addresses of a server whose id is 1, you'd do it in the following way:

```
$ curl -v -H "X-Auth-Token:999888777666" http://localhost:8774/v1.1/openstack/
servers/1/ips
```

What you will get in return will be a JSON response similar to the one below:

```
{
  "addresses": {
    "public": [
      {
        "version": 4,
        "addr": "192.168.100.2"
      },
      {
        "version": 6,
        "addr": "fe80:0:0:0:0:0:c0a8:6402"
      }
    ],
```

```
    "private": [
      {
        "version": 4,
        "addr": "192.168.200.2"
      },
      {
        "version": 6,
        "addr": "fe80:0:0:0:0:0:c0a8:6402"
      }
    ]
  }
}
```

The addresses dictionary contains two lists: private and public. These lists hold IPv4 and IPv6 addresses assigned to the server on private and public networks. The number of addresses asigned to your server may vary, but there should be at least one private IPv4 and one public IPv4 address.

Listing Server Addresses Using Python

When you want to extract the list of IP addresses for a server using Python, use the following recipe as your starting point:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Get the list of server IP addresses
###

# HTTP connection #2

srvID = 1       # server ID number
url2 = apiurlt[1]
params2 = urllib.urlencode({})
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("GET", "%s/servers/%s/ips" % (apiurlt[2], srvID), params2,
 headers2)
```

```
# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()

print dd2
```

If you want to access a specific address you need to use a dictionary/list access notation, e.g. if you want to print the first public IPv4 address:

```
ipAddr = dd2["addresses"]["public"][0]

if (ipAddr["version"] == 4):
    print ipAddr["addr"]
```

Listing Network-Specific Server Addresses

It is possible to request just private or public server addresses. You can list them using a GET request for the /servers/id/ips/network URL. For example, if you were to use curl to list the public IP addresses of a server whose id is 1, you'd do it in the following way:

```
$ curl -v -H "X-Auth-Token:999888777666" http://localhost:8774/v1.1/openstack/
servers/1/ips/public
```

What you will get in return will be a JSON response similar to the one below:

```
{
  "addresses": {
    "public": [
      {
        "version": 4,
        "addr": "192.168.100.2"
      },
      {
        "version": 6,
        "addr": "fe80:0:0:0:0:0:c0a8:6402"
      }
    ]
  }
}
```

Listing Network-Specific Server Addresses Using Python

Network-specific IP addresses can be retrieved using a Python script:

```
#!/usr/bin/python
```

```
import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Get the list of server IP addresses for a specific network
###

# HTTP connection #2

srvID = 1       # server ID number
networkName = "public"  # can be either "public" or "private"
url2 = apiurlt[1]
params2 = urllib.urlencode({})
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("GET", "%s/servers/%s/ips/%s" % (apiurlt[2], srvID,
 networkName), params2, headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()

print dd2
```

第 4 章 Server Metadata

Server metadata is stored in the metadata dictionary within the server description data structures. We already discussed adding metadata during server creation, now we'll see how to list, add, update, and delete metadata items associated with your servers.

Listing Server Metadata

It is possible to access server metadata information without having to list all properties of the server. For example, if you want to use curl to list metadata for a server whose id is 1, you can do it in the following way:

```
$ curl -H "X-Auth-Token:999888777666" http://localhost:8774/v1.1/openstack/
servers/1/metadata
```

The result should look similar to the output shown below:

```
{
  "metadata": {
    "Server Name": "Tornado"
  }
}
```

Notice that the GET method must be applied to the URL that ends with the /servers/id/metadata path.

Lisitng Server Metadata Using Python

When you'd rather use Python to access metadata information, you could do it in the following way:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Get server metadata
###
```

```
# HTTP connection #2

servID = 1      # server ID
url2 = apiurlt[1]
params2 = urllib.urlencode({})
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("GET", "%s/servers/%s/metadata" % (apiurlt[2], servID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()

print dd2
```

Listing Server Metadata Items

Using the /servers/id/metadata URI gets you the dictionary of all metadata keys. If you wanted to access a specific key, you can use a four-part URI, /servers/id/metadata/key, where id is the numeric id of the server and key is the metadata key you want the value of.

For example, if you want to get the value of the owner metadata key for server whose id is 1, you can do it using curl in the following way:

```
$ curl -H "X-Auth-Token:999888777666" http://localhost:8774/v1.1/openstack/
servers/1/metadata/owner
```

The result should look similar to the output shown below:

```
{
  "metadata": {
    "owner": "joe"
  }
}
```

Listing Server Metadata Items Using Python

If you wanted to check for a particular metadata key, you can reuse the Python script from the previous section. You already have the metadata dictionary, and all that's left is retrieving the key you are looking for:

```
print dd2['metadata']['owner']
```

Setting Server Metadata

Server metadata is stored in the metadata dictionary, which is a part of each image description. It doesn't have to be set to anything, but it is often used to add a more detailed description of the image's purpose than a terse name of the image.

Server metadata can be set at server creation time or later. When you want to do it after you create your server, you need to use the PUT method and a URL with a three-part path, /servers/id/metadata.

The id part is the numeric id of the server. For example, if you wanted to set the owner metadata key of the server whose id is 1, the path would be /servers/1/metadata.

If you were to use curl to set metadata keys, this is how you'd do it:

```
$ curl -X PUT -H "X-Auth-Token:999888777666" -H "Accept: application/json"
 -d '{"metadata":{"owner":"joe","group":"www"}}' http://localhost:8774/v1.1/
openstack/servers/1/metadata
```

Setting Server Metadata Using Python

When you want to use Python to set server metadata, you could do it in the following way:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Set server metadata
###

# HTTP connection #2

servID = 1      # server ID
url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"metadata":{"owner":"joe",
"group":"www"}}))
headers2 = { "X-Auth-Token":apitoken, "Accept":"application/json", "Content-
type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
```

```
    conn2 = httplib.HTTPConnection(url2)

conn2.request("PUT", "%s/servers/%s/metadata" % (apiurlt[2], servID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()
```

Setting Server Metadata Items

In the previous section you learned how to set multiple metadata items in one go. But what if you wanted to set them one at a time?

Once again, you need to use the PUT method and a URI that points to the metadata you wish to set, e.g. /servers/id/metadata/key.

The id part is the numeric id of the server. For example, if you wanted to set the owner metadata key of the server whose id is 1, the path would be /servers/1/metadata/owner.

The value of the key set has to be passed to the OpenStack Compute API server as a single-key metadata dictionary:

```
{
  "metadata": {
    "owner": "joe"
  }
}
```

If you were to use curl to set metadata keys, this is how you'd do it:

```
$ curl -X PUT -H "X-Auth-Token:999888777666" -H "Accept: application/json" -d
 '{"metadata":{"owner":"joe"}}' http://localhost:8774/v1.1/openstack/servers/
1/metadata/owner
```

Setting Server Metadata Items Using Python

When you want to use Python to set server metadata items, you can do it in the following way:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
```

```
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Set server metadata
###

# HTTP connection #2

servID = 1      # server ID
metaKey = "owner"   # metadata key
url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"metadata":{metaKey:"joe"}}))
headers2 = { "X-Auth-Token":apitoken, "Accept":"application/json", "Content-
type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("PUT", "%s/servers/%s/metadata/%s" % (apiurlt[2], servID,
 metaKey), params2, headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()
```

Updating Server Metadata

If you want to update an existing metadata key, you need to use the POST method and the same, three-part URL path, /servers/id/metadata.

The id part is the numeric id of the server. For example, if you wanted to set the owner metadata key of the server whose id is 1, the path would be /servers/1/metadata.

If you were to use curl to set metadata keys, this is how you'd do it:

```
$ curl -X POST -H "X-Auth-Token:999888777666" -H "Accept: application/json" -d
 '{"metadata":{"owner":"adam"}}' http://localhost:8774/v1.1/openstack/servers/
1/metadata
```

Updating Server Metadata Using Python

When you want to use Python to update server metadata, you could do it in the following way:

```
#!/usr/bin/python
```

```
import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Update server metadata
###

# HTTP connection #2

servID = 1      # server ID
url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"metadata":{"owner":"adam"}}))
headers2 = { "X-Auth-Token":apitoken, "Accept":"application/json", "Content-
type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("POST", "%s/servers/%s/metadata" % (apiurlt[2], servID),
 params2, headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()
```

Deleting Server Metadata Items

When you want to delete an existing metadata item, you will need to use the DELETE method and the four-part URI, /servers/id/metadata/key.

The id part is the numeric id of the server. For example, if you wanted to set the owner metadata key of the server whose id is 1, the path would be /servers/1/metadata/owner.

If you were to use curl to set metadata keys, this is how you'd do it:

```
$ curl -X DELETE -H "X-Auth-Token:999888777666" -H "Accept: application/json"
 http://localhost:8774/v1.1/openstack/servers/1/metadata/owner
```

Please note there is no data payload in a delete operation. It is a common mistake to leave it in while editing scripts.

Deleting Server Metadata Items Using Python

Deleting server metadata items using Python can be done in the way shown below:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Deleting server metadata items
###

# HTTP connection #2

servID = 1      # server ID
metaKey = "owner"   # metadata key
url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"metadata":{"owner":"adam"}}))
headers2 = { "X-Auth-Token":apitoken, "Accept":"application/json", "Content-
type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("DELETE", "%s/servers/%s/metadata/%s" % (apiurlt[2], servID,
 metaKey), params2, headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()
```

第 5 章 Server Image Metadata

Server image metadata is not the same as the server image data, but the way you use it, modify it, or delete it is very similar.

Server image metadata is stored in the metadata dictionary within the server image description data structures. We already discussed adding metadata during server image ### FIXME ### creation, now we'll see how to list, add, update, and delete metadata items associated with your server images.

Listing Server Image Metadata

It is possible to access server image metadata information without having to list all properties of the server image. For example, if you want to use curl to list metadata for a server image whose id is 1, you can do it in the following way:

```
$ curl -H "X-Auth-Token:999888777666" http://localhost:8774/v1.1/openstack/
images/1/metadata
```

The result should look similar to the output shown below:

```
{
  "metadata": {
    "imageName": "App Server"
  }
}
```

Notice that the GET method must be applied to the URL that ends with the /images/id/metadata path.

Lisitng Server Image Metadata Using Python

When you'd rather use Python to access metadata information, you could do it in the following way:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Get server image metadata
###
```

```
# HTTP connection #2

imgID = 1       # server image ID
url2 = apiurlt[1]
params2 = urllib.urlencode({})
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("GET", "%s/images/%s/metadata" % (apiurlt[2], imgID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()

print dd2
```

Listing Server Image Metadata Items

Using the /images/id/metadata URI gets you the dictionary of all metadata keys. If you wanted to access a specific key, you can use a four-part URI, /images/id/metadata/key, where id is the numeric id of the server image and key is the metadata key you want the value of.

For example, if you want to get the value of the owner metadata key for server image whose id is 1, you can do it using curl in the following way:

```
$ curl -H "X-Auth-Token:999888777666" http://localhost:8774/v1.1/openstack/
images/1/metadata/owner
```

The result should look similar to the output shown below:

```
{
  "metadata": {
    "owner": "joe"
  }
}
```

Listing Server Metadata Items Using Python

If you wanted to check for a particular metadata key, you can reuse the Python script from the previous section. You already have the metadata dictionary, and all that's left is retrieving the key you are looking for:

```
print dd2['metadata']['owner']
```

Setting Server Image Metadata

Server image metadata is stored in the metadata dictionary, which is a part of each image description. It doesn't have to be set to anything, but it is often used to add a more detailed description of the image's purpose than a terse name of the image.

Server image metadata can be set at server image creation time or later. When you want to do it after you create your server image, you need to use the PUT method and a URL with a three-part path, /images/id/metadata.

The id part is the numeric id of the server image. For example, if you wanted to set the owner metadata key of the server image whose id is 1, the path would be /images/1/metadata.

If you were to use curl to set metadata keys, this is how you'd do it:

```
$ curl -X PUT -H "X-Auth-Token:999888777666" -H "Accept: application/json"
 -d '{"metadata":{"owner":"joe","group":"www"}}' http://localhost:8774/v1.1/
openstack/images/1/metadata
```

Setting Server Image Metadata Using Python

When you want to use Python to set server image metadata, you could do it in the following way:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Set server image metadata
###

# HTTP connection #2

imgID = 1       # server image ID
url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"metadata":{"owner":"joe",
"group":"www"}}))
headers2 = { "X-Auth-Token":apitoken, "Accept":"application/json", "Content-
type":"application/json" }

if (usehttps == True):
```

```
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("PUT", "%s/images/%s/metadata" % (apiurlt[2], imgID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()
```

Setting Server Image Metadata Items

In the previous section you learned how to set multiple metadata items in one go. But what if you wanted to set them one at a time?

Once again, you need to use the PUT method and a URI that points to the metadata you wish to set, e.g. /images/id/metadata/key.

The id part is the numeric id of the server image. For example, if you wanted to set the owner metadata key of the server image whose id is 1, the path would be /images/1/metadata/owner.

The value of the key set has to be passed to the OpenStack Compute API server as a single-key metadata dictionary:

```
{
  "metadata": {
    "owner": "joe"
  }
}
```

If you were to use curl to set metadata keys, this is how you'd do it:

```
$ curl -X PUT -H "X-Auth-Token:999888777666" -H "Accept: application/json" -d
 '{"metadata":{"owner":"joe"}}' http://localhost:8774/v1.1/openstack/images/1/
metadata/owner
```

Setting Server Image Metadata Items Using Python

When you want to use Python to set server image metadata items, you can do it in the following way:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
```

```
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Set server image metadata
###

# HTTP connection #2

imgID = 1       # server image ID
metaKey = "owner"   # metadata key
url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"metadata":{metaKey:"joe"}}))
headers2 = { "X-Auth-Token":apitoken, "Accept":"application/json", "Content-
type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("PUT", "%s/image/%s/metadata/%s" % (apiurlt[2], imgID, metaKey),
 params2, headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()
```

Updating Server Image Metadata

If you want to update an existing metadata key, you need to use the POST method and the same, three-part URL path, /images/id/metadata.

The id part is the numeric id of the server image. For example, if you wanted to set the owner metadata key of the server image whose id is 1, the path would be /images/1/metadata.

If you were to use curl to set metadata keys, this is how you'd do it:

```
$ curl -X POST -H "X-Auth-Token:999888777666" -H "Accept: application/json" -d
 '{"metadata":{"owner":"adam"}}' http://localhost:8774/v1.1/openstack/images/
1/metadata
```

Updating Server Image Metadata Using Python

When you want to use Python to update server image metadata, you could do it in the following way:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Update server image metadata
###

# HTTP connection #2

imgID = 1       # server image ID
url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"metadata":{"owner":"adam"}}))
headers2 = { "X-Auth-Token":apitoken, "Accept":"application/json", "Content-
type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("POST", "%s/images/%s/metadata" % (apiurlt[2], imgID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()
```

Deleting Server Image Metadata Items

When you want to delete an existing metadata item, you will need to use the DELETE method and the four-part URI, /images/id/metadata/key.

The id part is the numeric id of the server image. For example, if you wanted to set the owner metadata key of the server image whose id is 1, the path would be /images/1/metadata/owner.

If you were to use curl to set metadata keys, this is how you'd do it:

```
$ curl -X DELETE -H "X-Auth-Token:999888777666" -H "Accept: application/json"
 http://localhost:8774/v1.1/openstack/images/1/metadata/owner
```

Please note there is no data payload in a delete operation. It is a common mistake to leave it in while editing scripts.

Deleting Server Image Metadata Items Using Python

Deleting server image metadata items using Python can be done in the way shown below:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Deleting server image metadata items
###

# HTTP connection #2

imgID = 1       # server image ID
metaKey = "owner"   # metadata key
url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"metadata":{"owner":"adam"}}))
headers2 = { "X-Auth-Token":apitoken, "Accept":"application/json", "Content-
type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("DELETE", "%s/images/%s/metadata/%s" % (apiurlt[2], imgID,
 metaKey), params2, headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()
```

Updating Servers: Changing Server Name

Contrary to what some users of cloud server think, it is not necessary to delete a server and create a new one when you want to change something about it. Some small modifications,

like changing server name or server access address can be done without such drastic measures.

If you want to change the server name, you will need to use a PUT request directed at /servers/id, where id is the numeric ID of the server you wish to update. You need to include a JSON payload:

```
{
  "server" : {
    "name" : "nginx000"
  }
}
```

If you were to use curl to change the server name, this is how you can do it:

```
$ curl -X PUT -H "X-Auth-Token:999888777666" -H "Accept: application/json"
 -d '{"server": {"name": "nginx000"}}' http://localhost:8774/v1.1/openstack/
servers/1
```

The response to this request will be a long JSON string similar to the one you get when you create a new server.

Updating Servers: Changing Server Name Using Python

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Changing server name
###

# HTTP connection #2

imgID = 1       # server image ID
srvName = "nginx000"    # server name
url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"server": {"name":srvName}}))
headers2 = { "X-Auth-Token":apitoken, "Accept":"application/json", "Content-
type":"application/json" }

if (usehttps == True):
```

```
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("DELETE", "%s/servers/%s" % (apiurlt[2], imgID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()

print dd2
```

Updating Servers: Changing Server Access Address

If you want to change server access IPv4 and/or IPv6, you can do it using a PUT request directed at /servers/id, where id is the numeric ID of the server you wish to update. You need to include a JSON payload:

```
{
  "server" : {
    "accessIPv4" : "192.168.100.2",
    "accessIPv6" : "fe80:0:0:0:0:0:c0a8:6402"
  }
}
```

If you were to use curl to change the server access addresses, this is how you can do it:

```
$ curl -X PUT -H "X-Auth-Token:999888777666" -H "Accept: application/
json" -d '{"server": {"accessIPv4": "192.168.100.2", "accessIPv6":
 "fe80:0:0:0:0:0:c0a8:6402"}}' http://localhost:8774/v1.1/openstack/servers/1
```

The response to this request will be a long JSON string similar to the one you get when you create a new server.

Updating Servers: Changing Server Access Address Using Python

If you want to use Python to change server access IP addresses, you could start with the following script:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse
```

```
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Changing server name
###

# HTTP connection #2

imgID = 1               # server image ID
srvIPv4 = "192.168.100.2"       # server access IPv4
srvIPv6 = "fe80:0:0:0:0:0:c0a8:6402"    # server access IPv6
url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"server": {"accessIPv4": srvIPv4,
 "accessIPv6": srvIPv6}}))
headers2 = { "X-Auth-Token":apitoken, "Accept":"application/json", "Content-
type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("DELETE", "%s/servers/%s" % (apiurlt[2], imgID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()

print dd2
```

Changing Root (Administrator) Password

The root (Administrator) password is the most important of all passwords on any server. You should never log in as root or Administrator, but you sometimes have to do it for basic initial administrative tasks like adding users, setting up sudo priviledges, etc. Your cloud infrastructure provider will most likely provide a web interface for changing the root password, but you can also do it via the OpenStack Compute API.

What you need to do is use a POST request directed at /servers/id/action, where id is the numeric server ID. You also need to include a payload with the description of the action you want to perform on your server (changePassword), the parameters of those actions, and their values. It has to be a JSON dictionary structure:

```
{
  "changePassword" : {
    "adminPass" : "dontusesshhhforpassword"
```

```
  }
}
```

If you want to use curl to reset root or Administrator, here's how you can do it:

```
$ curl -s -X POST -H "X-Auth-Token:999888777666" -d '{"changePassword":
 {"adminPass": "dontusesshhhforpassword"}}' http://localhost:8774/v1.1/
openstack/servers/3/action
```

Changing Root (Administrator) Password Using Python

If you want to delete a server using Python, here's how it can be done:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Get the list of server IP addresses for a specific network
###

# HTTP connection #2

srvID = 1               # server ID number
passStr = "dontusesshhhforpassword"

url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"changePassword": {"adminPass":
 passStr}}))
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("POST", "%s/servers/%s/action" % (apiurlt[2], srvID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
```

```
conn2.close()
```

Rebooting Servers

One of the most common administrative tasks is server reboot. You can log into a server as root (Administrator) and use the shutdown command:

```
# shutdown -r now
```

But what if you cannot log into the server or when you have a couple of dozens of servers to reboot? The answer is a POST request directed at /servers/id/action, where id is the numeric server ID. You also need to include a payload with the description of the action you want to perform on your server (reboot), the parameters of those actions, and their values. It has to be a JSON string:

```
{
  "reboot" : {
    "type" : "HARD"
  }
}
```

There are two types of reboot:

- HARD — the equivalent of turning power off and on again.
- SOFT — a gentle kind of reboot, think shutdown -r now.

If you want to use curl to reboot a server, you can do it in the following way:

```
$ curl -s -X POST -H "X-Auth-Token:999888777666" -d '{"reboot": {"type":
 "HARD"}}' http://localhost:8774/v1.1/openstack/servers/3/action
```

Rebooting Servers Using Python

If you want to reboot a server using Python, here's how it can be done:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Get the list of server IP addresses for a specific network
###
```

```
# HTTP connection #2

srvID = 1       # server ID number
rebootType = "HARD" # can be either "SOFT" or "HARD"

url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"reboot": {"type": rebootType}}))
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("POST", "%s/servers/%s/action" % (apiurlt[2], srvID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
conn2.close()
```

Rebuilding Servers

When things go wrong, e.g. you log in as root and overwrite /bin, you can simply rebuild the server. This operation will delete all data and install a fresh copy of the operating system, which does not have to be the system you used to create the original server. What you get after rebuilding a server is a new machine with the old machine's reference URL and IP addresses.

The POST request sent to /servers/id/action should carry a JSON payload that includes the rebuild command, server image reference, name, root/(Administration) password, server metadata, and personalization files. The id is the numeric ID of the server you want to renuild.

You do not need to provide the flavor reference, because the virtual hardware your rebuilt server will run on does not change. If you want to alter that, rebuild your server and resize it afterwards (see the next section).

The following script demonstrates how to rebuild a server using Python. I skipped the metadata and personalization setup for clarity, but you can find out how to use it if you read earlier sections on that subject.

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
```

```
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Get server image reference
###

# HTTP connection #2

url2 = apiurlt[1]
params2 = urllib.urlencode({})
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("GET", "%s/images" % apiurlt[2], params2, headers2)

# HTTP response

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()

###
### Server parameters
###

# Server name

sname = "tornado001"

# Server image URL

n = len(dd2["images"])
m = range(n)

for i in m:
    if dd2["images"][i]["id"] == 1:
        sImageRef = dd2["images"][i]["links"][0]["href"]

###
### server metadata
###

sMetadata = {}

###
### server personalization
###

sPersonalityPath = ""
sPersonalityContents = ""
```

```
sPersonality = [ { "path":sPersonalityPath, "contents":base64.
b64encode( sPersonalityContents ) } ]

s = { "rebuild": { "name": sname, "imageRef": sImageRef, "metadata":
 sMetadata, "personality": sPersonality } }

sj = json.dumps(s)

# HTTP connection #3

params3 = sj
headers3 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

conn3 = httplib.HTTPConnection("localhost:8774")
conn3.request("POST", "%s/servers" % apiurlt[2], params3, headers3)

# HTTP response #3

response3 = conn3.getresponse()
data3 = response3.read()
dd3 = json.loads(data3)

conn3.close()

print json.dumps(dd3, indent=2)
```

Resizing Servers

When you need more memory or more storage space (or both), you can resize the virtual hardware your server runs on.

The POST request sent to /servers/id/action must carry a simple JSON payload with the name of the operation, resize, and the new server flavor URL like the one below (it points to the server flavor number 5):

```
{
  "resize": {
    "flavorRef": "http://localhost:8774/v1.1/openstack/flavors/5"
  }
}
```

If you want to use curl, the server resize request would look like this:

```
$ curl -s -X POST -H "X-Auth-Token:999888777666" -d '{"resize": {"flavorRef":
 "http://localhost:8774/v1.1/openstack/flavors/5"}}' http://localhost:8774/v1.
1/openstack/servers/3/action
```

It is possible to resize your server down, but it is up to you to make sure that your data fits on the new, smaller disks.

Resizing Servers Using Python

The following example shows how to resize a server using Python. It looks more complex than the curl example, but that's only because it also shows how to exrtact the flavor URL:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Resize a server
###

# HTTP connection #2

srvID = 1       # server ID number
flavorID = 3        # flavor ID number

url2 = apiurlt[1]

# Flavor URL

# HTTP connection #2

params2 = urllib.urlencode({})
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("GET", "%s/flavors" % apiurlt[2], params2, headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
dd2 = json.loads(data2)

conn2.close()

n = len(dd2["flavors"])
m = range(n)

for i in m:
    if dd2["flavors"][i]["id"] == flavorID:
        sFlavorRef = dd2["flavors"][i]["links"][0]["href"]

# HTTP connection #3

params3 = urllib.urlencode(json.dumps({"resize": {"flavorRef": sFlavorRef}}))
headers3 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }
```

```
if (usehttps == True):
    conn3 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn3 = httplib.HTTPConnection(url2)

conn3.request("POST", "%s/servers/%s/action" % (apiurlt[2], srvID), params3,
 headers3)

# HTTP response #3

response3 = conn3.getresponse()
data3 = response3.read()
conn3.close()
```

Confirming Server Resize

The OpenStack Compute cloud will wait for 24 hours before making changes permanent. This gives you a chance to test things and go back to the old flavor if needed. It is up to you if you want to confirm the resize, the OpenStack Compute cloud will do it anyway after 24 hours, but if you are happy, you can do it by sending a POST request to /servers/id/action. The id parameter is the numeric ID of the server you resized.

Don't forget the JSON payload:

```
{
  "confirmResize": null
}
```

If you want to use curl, the server resize request would look like this:

```
$ curl -s -X POST -H "X-Auth-Token:999888777666" -d '{"confirmResize": null}'
 http://localhost:8774/v1.1/openstack/servers/3/action
```

Confirming Server Resize Using Python

If you want to use Python to confirm server resize, use this as your inspiration:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
```

```
###
### Confirm server resize
###

# HTTP connection #2

srvID = 1       # server ID number

url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"confirmResize": null}))
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("POST", "%s/servers/%s/action" % (apiurlt[2], srvID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
conn2.close()
```

Reverting Server Resize

If you are not happy with your reszied server, you can revert to the old flavor using the revertResize command.

You can do it by sending a POST request to /servers/id/action. The id parameter is the numeric ID of the server you resized.

Don't forget the JSON payload:

```
{
  "revertResize": null
}
```

If you want to use curl, the server resize request would look like this:

```
$ curl -s -X POST -H "X-Auth-Token:999888777666" -d '{"revertResize": null}'
 http://localhost:8774/v1.1/openstack/servers/3/action
```

Reverting Server Resize Using Python

If you want to use Python to revert server resize, use this as your inspiration:

```
#!/usr/bin/python

import base64
import urllib
```

```
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Revert server resize
###

# HTTP connection #2

srvID = 1       # server ID number

url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"revertResize": null}))
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("POST", "%s/servers/%s/action" % (apiurlt[2], srvID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
conn2.close()
```

第 6 章 Server Images

Servers can be created using default templates, but you can also create your own server images that will be later reused to quickly replicate your working setup. Server images preserve the customiztions and the data from the original server. This allows you to pre-load one server with data and replicate it to divide the data processing job between as many machines as you need.

Creating Server Images

The server image creation operation is quite simple, all you need to do is send a POST request to /servers/id/action. The id is the numeric ID of the server that you want to preserve.

The JSON payload that accompanies the createImage command must include the name of the server image and optional metadata. That metadata is independent of the server metadata.

```
{
  "createImage": {
    "name": "tornado-app-server",
    "metadata": {
      "version": "1.0.0",
      "creator": "joe"
    }
      }
}
```

If you want to use curl, the server image creation request would look like this:

```
$ curl -s -X POST -H "X-Auth-Token:999888777666" -d '{"createImage": {"name":
 "tornado-app-server", "metadata": {"version": "1.0.0", "creator": "joe"}}}'
 http://localhost:8774/v1.1/openstack/servers/3/action
```

Creating Server Images Using Python

If you want to use Python to to create a server image, use this as your inspiration:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
```

```
###
### Revert server resize
###

# HTTP connection #2

srvID = 1               # server ID number
imgName = "tornado-app-server"
metaDict = {"version": "1.0.0", "creator": "joe"}

url2 = apiurlt[1]
params2 = urllib.urlencode(json.dumps({"createImage": {"name": imgName,
 "metadata": metaDict}}))
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("POST", "%s/servers/%s/action" % (apiurlt[2], srvID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
conn2.close()
```

Deleting Servers

When a server is no longer necessary you can simply delete it. One REST call and the server is gone. You will need to add the server id number after /servers and you must use the DELETE method, as in:

```
$ curl -s -X DELETE -H "X-Auth-Token:999888777666" http://localhost:8774/v1.1/
openstack/servers/3
```

The OpenStack Compute API server will not return a JSON object when the operation is successful, but it will return an error message when the server you are trying to delete does not exist:

```
{
  "itemNotFound": {
    "message": "The resource could not be found.",
    "code": 404
  }
}
```

Deleting Servers Using Python

When you use Python, server deletion procedure can be done in a fairly simple way:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Get the list of server IP addresses for a specific network
###

# HTTP connection #2

srvID = 1       # server ID number
url2 = apiurlt[1]
params2 = urllib.urlencode({})
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("GET", "%s/servers/%s/delete" % (apiurlt[2], srvID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
conn2.close()
```

Deleting Server Images

When a server image is no longer necessary you can get rid of it. You will need to send a DELETE request to /servers/id, where id is the numeric ID of the server image you want to remove:

```
$ curl -s -X DELETE -H "X-Auth-Token:999888777666" http://localhost:8774/v1.1/
openstack/images/3
```

The OpenStack Compute API server will not return a JSON object when the operation is successful, but it will return an error message when the server you are trying to delete does not exist:

```
{
  "itemNotFound": {
    "message": "The resource could not be found.",
```

```
    "code": 404
  }
}
```

Deleting Servers Images Using Python

When you use Python, server image deletion procedure can be done in a fairly simple way:

```
#!/usr/bin/python

import base64
import urllib
import httplib
import json
import os
from urlparse import urlparse

### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--
###
###  insert the 'Get OpenStack Credentials' snippet here
###
### --8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--8<--

###
### Delete server image
###

# HTTP connection #2

srvID = 1       # server ID number
url2 = apiurlt[1]
params2 = urllib.urlencode({})
headers2 = { "X-Auth-Token":apitoken, "Content-type":"application/json" }

if (usehttps == True):
    conn2 = httplib.HTTPSConnection(url2, key_file='../cert/priv.pem',
 cert_file='../cert/srv_test.crt')
else:
    conn2 = httplib.HTTPConnection(url2)

conn2.request("DELETE", "%s/images/%s" % (apiurlt[2], srvID), params2,
 headers2)

# HTTP response #2

response2 = conn2.getresponse()
data2 = response2.read()
conn2.close()
```

第 7 章 Additional Tools

A while ago I wrote a simple tool for pretty-printing JSON data. Over time I found it handy to have around when working with various applications and services that generate JSON output. You might find it handy when you are working with JSON. Here it is:

```
#!/usr/bin/python

import sys
import json

def main():

        try:

                data = sys.stdin.readline()
                dd = json.loads(data)

        except:

                sys.stderr.write("Error: %s" % str(sys.exc_info()[0]))
                sys.stderr.write("\nI got no data, badly formatted JSON, or
 something that is not JSON.")

                sys.exit()

        print json.dumps(dd, indent=2)

if __name__ == "__main__":
    main()
```