Python APIs: The best-kept secret of OpenStack

Writing OpenStack automation scripts with Python bindings

Lorin Hochstein June 19, 2013

As an OpenStack user or administrator, you often need to write scripts to automate common tasks. In addition to the REST and command-line interfaces, OpenStack exposes native Python API bindings. Learn how to use these Python bindings to greatly simplify the process of writing OpenStack automation scripts.

OpenStack is an increasingly popular open source solution for deploying Infrastructure as a Service (IaaS) clouds. OpenStack ships with a dashboard web app that works well for performing manual tasks, such as launching a single virtual machine (VM) instance, but if you want to automate your cloud-based tasks, you'll need to write scripts that can drive OpenStack.

Services in OpenStack

The term *service* in OpenStack is overloaded. It is used to refer to:

- An OpenStack project (for example, Compute Service (`nova`), Identity Service (`keystone`)
- An entry in the Identity Service catalog (for example, image, compute, volume)
- A Linux® daemon (for example, `nova-api`, `quantum-l3-agent`)

An OpenStack project is associated with one or more entries in the Identity Service catalog and is implemented by one or more Linux daemons. In the context of this article, I use service to mean OpenStack project.

Many users write either automation scripts directly against the OpenStack Representational State Transfer (REST) application programming interface (API) or shell scripts that invoke the command-line tools (for example, `keystone` or `nova`). But a better way exists to write OpenStack automation scripts in Python. All of the OpenStack services expose native Python APIs that expose the same feature set as the command-line tools. Unfortunately, not much documentation is available to describe how to use these APIs.

If you're a Python programmer, the Python APIs are much simpler to work with than command-line tools or the REST API. In this article, I demonstrate how you can use the native OpenStack Python APIs to automate common user and administrative tasks.

OpenStack projects and code names

The term *OpenStack* doesn't refer to a single application. Rather, it's a collection of services that work in concert to implement an IaaS cloud. (See the sidebar Services in OpenStack for what service means here.) Each OpenStack service has an official name and a code name, as shown in Table 1, and every OpenStack service exposes its own Python API.

OpenStack services and code names

Official name	Code name
Identity Service	`keystone`
Image Service	`glance`
Compute Service	`nova`
Networking Service	`quantum`
Block Storage Service	`cinder`
Object Storage Service	`swift`

Installing the Python bindings

The Python bindings are bundled with the command-line tools for each service. In fact, each command-line tool is implemented using the corresponding Python API. You can install each tool from the Python Package Index. The `pip` package names are:

- `python-keystoneclient`
- `python-glanceclient`
- `python-novaclient`
- `python-quantumclient`
- `python-cinderclient`
- `python-swiftclient`

For example, to install the `keystone` client, run the following command:

```
$ pip install python-keystoneclient
```

You can install these packages into a Python virtualenv or into your system-wide Python packages, if you have root privileges on your local machine.

All of the OpenStack APIs are versioned, and the Python bindings support multiple API versions to maintain backwards compatibility. As a result, it's safe to download the latest version of these packages, because they will work with all older versions of OpenStack services.

In this article, I focus on Python API examples from the following services:

- OpenStack Identity Service (`keystone`)
- OpenStack Image Service (`glance`)
- OpenStack Compute Service (`nova`)

Setting up a test environment

To get the most out of this article, I recommend that you have access to an OpenStack cloud with administrator privileges so that you can try the code snippets. If you don't currently have admin access to an OpenStack cloud, the simplest thing to do is deploy OpenStack inside a VM. The DevStack project was designed to makes it simple to create a development-oriented deployment of OpenStack on a single machine. Paired with a virtualization tool like VirtualBox, you can bring up an OpenStack cloud on your laptop—even on Mac or Windows®.

You can also get a free account on TryStack, the community-maintained OpenStack sandbox. Note that you can only get user-level privileges on TryStack, not admin-level privileges, so you won't be able to use TryStack to test any scripts that require admin privileges.

OpenStack Identity (keystone)

A client makes requests against the Identity (`keystone`) API by instantiating the appropriate `keystone` client Python object and calling its methods. Because the APIs are versioned, the Python clients are always associated with a specific version of the API.

Listing 1 shows an example of using version 2.0 of the `keystone` client to add the Image Service to the service catalog.

Creating an admin role with keystone

```
import keystoneclient.v2_0.client as ksclient
# Replace the method arguments with the ones from your local config
keystone = ksclient.Client(auth_url="http://192.168.27.100:35357/v2.0",
                           username="admin",
                           password="devstack",
                           tenant_name="demo")
glance_service = keystone.services.create(name="glance",
                            service_type="image",
                            description="OpenStack Image Service")
```

Credentials

You must supply credentials when instantiating a `keystoneclient.v2_0.client.Client` object. A `keystone` endpoint accepts two types of credentials: a token or a user name and password. If you are an administrator, you can use the `admin` token, which is a special token that has administrator privileges and never expires. You define this token with the `admin_token` configuration option in the /etc/keystone/keystone.conf file on the machine that runs the `keystone` service (see Listing 2).

Authenticating with the auth token

```
import keystoneclient.v2_0.client as ksclient

# Replace the values below with the ones from your local config
endpoint = "http://192.168.27.100:35357/v2.0"
admin_token = "devstack"

keystone = ksclient.Client(endpoint=endpoint, token=admin_token)
```

Using the `admin` token is generally frowned upon for security reasons. Instead, the OpenStack Identity developers recommend that you always use a user name and password for authentication after you have created a user who has admin privileges (see Listing 3).

Authenticating with a user name and password

```
import keystoneclient.v2_0.client as ksclient

# Replace the values below  the ones from your local config,
auth_url = "http://192.168.27.100:35357/v2.0"
username = "admin"
password = "devstack"
tenant_name = "demo"

keystone = ksclient.Client(auth_url=auth_url, username=username,
                           password=password, tenant_name=tenant_name)
```

Loading an openrc file

To simplify authentication, I recommend that you create an openrc file that exports the credentials to environment variables. Doing so allows you to avoid hard-coding login information into your scripts. Listing 4 shows an example openrc file.

Loading credentials from environment variables

```
export OS_USERNAME="myname"
export OS_PASSWORD="mypassword"
export OS_TENANT_NAME="mytenant"
export OS_AUTH_URL="http://10.20.0.2:5000/v2.0/"
```

The environment variables OS_USERNAME, OS_PASSWORD, OS_TENANT_NAME, and OS_AUTH_URL are standardized across all of the Python command-line tools. If these environment variables are set, the command-line tools (`keystone`, `nova`) will use them to authenticate against their API endpoints.

Load these environment variables into your current shell with the Bash `source` built-in command. If you use Bash as your standard shell, you may want to add this line to your .profile so that the environment variables are automatically set each time you log in:

```
$ source openrc
```

When the openrc file has been sourced, Python scripts can retrieve the credentials from the environment. Let's create a Python file called credentials.py, as shown in Listing 5, that extracts the login information from the environment. Note that `keystone` and `nova` use slightly different variable names in their client initializer methods, so I define separate functions for each.

credentials.py

```
#!/usr/bin/env python
import os

def get_keystone_creds():
    d = {}
    d['username'] = os.environ['OS_USERNAME']
    d['password'] = os.environ['OS_PASSWORD']
    d['auth_url'] = os.environ['OS_AUTH_URL']
    d['tenant_name'] = os.environ['OS_TENANT_NAME']
    return d

def get_nova_creds():
    d = {}
    d['username'] = os.environ['OS_USERNAME']
    d['api_key'] = os.environ['OS_PASSWORD']
    d['auth_url'] = os.environ['OS_AUTH_URL']
    d['project_id'] = os.environ['OS_TENANT_NAME']
    return d
```

Authentication tokens

If the client initializer returns without throwing an exception, it has successfully authenticated against the endpoint. You can access the `keystone` token that you were just issued through the `auth_token` attribute of the returned object, as shown in Listing 6. When authenticating against the `glance` API, you will need to explicitly pass a `keystone` authentication token as an argument to the initializer, as discussed later.

Successfully authenticating against a keystone endpoint in an interactive Python session

```
>>> import keystoneclient.v2_0.client as ksclient
>>> from credentials import get_keystone_creds
>>> creds = get_keystone_creds()
>>> keystone = ksclient.Client(**creds)
>>> keystone.auth_token
u'MIILkAYJKoZIhvcNAQcCoIILgTCCC30CAQExCTAHBgUrDgMCGjCCCmkGCSqGSIb3DQEHAaCCCloE
ggpWeyJhY2Nlc3MiOiB7InRva2VuIjogeyJpc3N1ZWRfYXQiOiAiMjAxMy0wNS0yNlQwMjoxMjo0Mi
42MDAwMjUiLCAiZXhwaXJlcyI6ICIyMDEzLTA1LTI3VDAyOjEyOjQyWiIsICJpZCI6ICJwbGFjZWhv
bGRlciIsICJ0ZW5hbnQiOiB7ImRlc2NyaXB0aW9uIjogbnVsbCwgImVuYWJsZWQiOiB0cnVlLCAiaW
...
fI9JnOBZJwuoma8je0a1AvLff6AcJ1zFkVZGb'
```

Note: The Grizzly release of OpenStack Identity uses Public Key Infrastructure tokens by default, which are much longer than the universally unique identifier tokens (for example, `7d9fde355f09458f8e97986a5a652bfe`) used in previous releases of OpenStack.

CRUD operations

The `keystone` API is essentially a create, read, update, delete (CRUD) interface: Most interactions with the `keystone` API either read from the `keystone` back-end database or modify it. Most interactions with the API happen by making calls on `Manager` objects. A Manager represents a collection of objects of the same type. For example, a `UserManager` manipulates `keystone` users, a `TenantManager` manipulates tenants, a `RoleManager` manipulates roles, and so on. The managers support operations such as `create` (create a new object), `get` (retrieve an object by ID), `list` (retrieve all of the objects), and `delete`.

Creating users, tenants, and roles

Typically, one of the first tasks you perform when deploying OpenStack is to create a `keystone` tenant, and then a `keystone` user with administrative privileges. Listing 7 shows an example of how to automate this process by using the Python API. The script performs the following tasks:

- Create a user role (`Client.roles.create`).
- Create an admin role (`Client.roles.create`).
- Create a tenant named acme (`Client.tenants.create`).
- Create a user named admin (`Client.users.create`).
- Assign the admin user the admin role in the acme tenant (`Client.roles.add_user_role`).

This is a good scenario for using the `admin` token, because the Identity Service does not yet contain any users with administrative privileges.

Creating a user, tenant, and role

```
import keystoneclient.v2_0.client as ksclient
endpoint = "http://192.168.27.100:35357/v2.0"
admin_token = "devstack"

keystone = ksclient.Client(endpoint=endpoint, token=admin_token)
user_role = keystone.roles.create("user")
admin_role = keystone.roles.create("admin")
acme_tenant = keystone.tenants.create(tenant_name="Acme",
                        description="Employees of Acme Corp.",
                        enabled=True)
admin_user = keystone.users.create(name="admin",
                password="a.G'03134!j",
                email="cloudmaster@example.com", tenant_id=acme_tenant.id)
keystone.roles.add_user_role(admin_user, admin_role, acme_tenant)
```

Creating services and endpoints

Typically, the next task in deploying the Identity Service in an OpenStack cloud is populating `keystone` with the services in the cloud and the endpoints. Listing 8 shows an example of how to add a service and an endpoint for the Identity Service using the `Client.services.create` and `Client.endpoints.create` methods.

Creating a service and endpoint

```
import keystoneclient.v2_0.client as ksclient
creds = get_keystone_creds() # See xxx
keystone = ksclient.Client(**creds)
service = keystone.services.create(name="keystone",
                                   service_type="identity",
                                   description="OpenStack Identity Service")

keystone_publicurl = "http://192.168.27.100:5000/v2.0"
keystone_adminurl = "http://192.168.27.100:35357/v2.0"
keystone.endpoints.create(service_id=service.id,
                          region="Northeast",
                          publicurl=keystone_publicurl,
                          adminurl=keystone_adminurl,
                          internalurl=keystone_publicurl)
```

Accessing the service catalog

One of `keystone`'s primary functions is to serve as a service catalog. Clients can use `keystone` to look up the endpoint URL of an OpenStack service. The API exposes this functionality through the `keystoneclient.v2_0.client.Client.service_catalog.url_for` method. This method allows you to look up a service endpoint by its type (for example, image, volume, compute, network) and its endpoint type (`publicURL`, `internalURL`, `adminURL`).

Listing 9 demonstrates how to use the `url_for` method to retrieve the endpoint of the OpenStack Image (`glance`) Service.

Querying for the glance endpoint in an interactive Python session

```
>>> import keystoneclient.v2_0.client as ksclient
>>> creds = get_keystone_creds() # See <a href="openrc-creds" />
>>> keystone = ksclient.Client(**creds)
>>> glance_endpoint = keystone.service_catalog.url_for(service_type='image',
                                                       endpoint_type='publicURL')
>>> glance_endpoint
u'http://192.168.27.100:9292'
```

OpenStack Compute (nova)

Version 1.1 vs. version 2 of the nova API

Version 1.1 and version 2 of the `nova` API are identical. You can pass "2" instead of "1.1" as the first argument of the `novaclient.client.Client initializer`, but no `novaclient.v2` module exists—only a `novaclient.v1_1` module.

The OpenStack Compute (`nova`) Python API works similarly to the OpenStack Identity API. I use version 1.1 of the `nova` API here, so the classes in version 1.1 of the `nova` Python bindings I use in this article are in the `novaclient.v1_1` Python namespace.

Authenticating against the nova-api endpoint

You make requests against the `nova-api` endpoint by instantiating a `novaclient.v1_1.client.Client` object and making calls against it. You can retrieve a client that communicates with version 1.1 of the API in two ways. Listing 10 demonstrates how to retrieve the appropriate client by passing the version string as an arguments.

Pass version as argument

```
from novaclient import client as novaclient
from credentials import get_nova_creds
creds = get_nova_creds()
nova = novaclient.Client("1.1", **creds)
```

Listing 11 demonstrates how to retrieve the appropriate client by importing the version 1.1 module explicitly.

Import version directly

```
import novaclient.v1_1.client as nvclient
from credentials import get_nova_creds
creds = get_nova_creds()
nova = nvclient.Client(**creds)
```

Listing instances

Use the `Client.servers.list` method to list the current VM instances, as shown in Listing 12.

Retrieve a list of VM instances in an interactive Python session

```
>>> import novaclient.v1_1.client as nvclient
>>> from credentials import get_nova_creds
>>> creds = get_nova_creds()
>>> nova = nvclient.Client(**creds)
>>> nova.servers.list()
[<Server: cirros>]
```

Retrieving an instance by name and shutting it down

If you don't know the ID of an instance but you know its name, you can use the `Server.find` method. Listing 13 shows how to find an instance by name and terminate it using the `Server.delete` method.

Terminate the "my-vm" instance

```
import novaclient.v1_1.client as nvclient
from credentials import get_nova_creds
creds = get_nova_creds()
nova = nvclient.Client(**creds)

server = nova.servers.find(name="my-vm")
server.delete()
```

Booting an instance and checking status

To start up a new instance, you use the `Client.servers.create` method, as shown in Listing 14. Note that you must pass an `image` object and `flavor` object, not the names of the image and flavor. The example also uses the `Client.keypairs.create` method to upload the Secure Shell (SSH) public key at `~/.ssh/id_rsa.pub` and names the key pair mykey, assuming that key pair does not yet exist. Finally, it uses the `Client.servers.get` method to retrieve the current state of the instance, which it uses to poll for status.

Booting a new instance

```
import os
import time
import novaclient.v1_1.client as nvclient
from credentials import get_nova_creds
creds = get_nova_creds()
nova = nvclient.Client(**creds)
if not nova.keypairs.findall(name="mykey"):
    with open(os.path.expanduser('~/.ssh/id_rsa.pub')) as fpubkey:
        nova.keypairs.create(name="mykey", public_key=fpubkey.read())
image = nova.images.find(name="cirros")
flavor = nova.flavors.find(name="m1.tiny")
instance = nova.servers.create(name="test", image=image, flavor=flavor, key_name="mykey")

# Poll at 5 second intervals, until the status is no longer 'BUILD'
status = instance.status
while status == 'BUILD':
    time.sleep(5)
```

```
    # Retrieve the instance again so the status field updates
    instance = nova.servers.get(instance.id)
    status = instance.status
print "status: %s" % status
```

Attaching a floating IP address

To attach a floating IP address, you must first verify that OpenStack has an available floating IP address. Use the `Client.floating_ips.list` method to retrieve a list of available floating IP addresses. If the resulting list is empty, allocate a new floating IP address by using the `Client.floating_ips.create` method, then assign it to the instance using the `Server.add_floating_ip` method, as shown in Listing 15.

Creating a floating IP address

```
>>> nova.floating_ips.list()
[]
>>> floating_ip = nova.floating_ips.create()
<FloatingIP fixed_ip=None, id=1, instance_id=None, ip=192.168.27.129, pool=public>
>>> instance = nova.servers.find(name="test")
>>> instance.add_floating_ip(floating_ip)
```

Changing a security group

Use the `Client.security_group_rules.create` method to add rules to a security group. Listing 16 shows an example that modifies the default security group to allow SSH (which runs on port 22) as well as all Internet Control Message Protocol (ICMP) traffic. To do so, I use the `Client.security_groups.find` method to retrieve the security group named default.

Allowing port 22 and ICMP in the default security group

```
import novaclient.v1_1.client as nvclient
from credentials import get_nova_creds
creds = get_nova_creds()
nova = nvclient.Client(**creds)

secgroup = nova.security_groups.find(name="default")
nova.security_group_rules.create(secgroup.id,
                                 ip_protocol="tcp",
                                 from_port=22,
                                 to_port=22)
nova.security_group_rules.create(secgroup.id,
                                 ip_protocol="icmp",
                                 from_port=-1,
                                 to_port=-1)
```

Retrieving the console log

The `Server.get_console_output` method retrieves the text sent to the console when the VM boots up. You can parse the console output to work around a common issue with changing host keys.

One of the drawbacks of an IaaS cloud such as OpenStack is that it doesn't interoperate well with SSH host key checking. If you are logging in to an instance at, say, 10.40.1.150, and that IP address had been previously used by another instance that you had logged in to in the past, you'll get an error that looks like Listing 17.

Error when host ID has changed

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
6f:2b:59:46:cb:8c:81:48:06:f3:c5:db:40:23:d3:be.
Please contact your system administrator.
Add correct host key in /home/mylogin/.ssh/known_hosts to get rid of this message.
Offending key in /home/mylogin/.ssh/known_hosts:1
RSA host key for 10.40.1.150 has changed and you have requested strict checking.
Host key verification failed.
```

If your VM image has the `cloud-init` package installed, then it will output the host key to the console, as shown in Listing 18.

Example SSH key output in console

```
-----BEGIN SSH HOST KEY KEYS-----
ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBDciNMyzj0osyPOM+
1OyseTWgkzw+M43zp5H2CchG8daRDHel7V3OHETVdI6WofNn
SdBJAwIoisRFPxyroNGiVw= root@my-name
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDU854+fNdcKMZTLCUejMOZllQmmphr6V5Aaz1F2+x2jXql5rqKQ
d5/h6OdFszcp+gdTeVtfgG++/298qodTemVVrvqwjp4eN87iHvhPxH6GDEevAKlEed2ckdAmgvzI9rcOYgR/46G9x
Iea0IdgNjMvN1baj6WPtv+HfcfH/ZV58G306lSJfbz/GVxNTIxW+Wg7ZQCAe6jWgm4oQ+66sco+7Fub24EPue3kO8
jqufqq3mY5+MFlzEHSX5B04ioG5Alw/JuqVx5+7zHt9I2wA3nzsyUdKtCTrw8V4fYEhWDm53WLOpW+8CeYCXuv+yL
7EjwLqhIH/TUuzGQiWmFGvyz root@my-name
-----END SSH HOST KEY KEYS-----
```

Listing 19 shows a script that uses the `Server.get_console_output` API method to extract the SSH host key from the console and update the `~/.ssh/known_hosts` file so you don't get that SSH warning when you use SSH to access the floating IP instance the first time.

Extracting the SSH host key from the console

```
import os
import subprocess
import novaclient.v1_1.client as nvclient
from credentials import get_nova_creds


def get_server(creds, servername):
    nova = nvclient.Client(**creds)
    return nova.servers.find(name=servername)


def remove_hostkey(ip):
    subprocess.call(["ssh-keygen", "-R", ip])


def get_hostkey_from_console(text):
    lines = text.split('\n')
    start = '-----BEGIN SSH HOST KEY KEYS-----\r'
    end = '-----END SSH HOST KEY KEYS-----\r'
    start_ind = lines.index(start)
    end_ind = lines.index(end)
    for i in range(start_ind+1, end_ind):
        key = lines[i].rstrip()
        if key.startswith('ssh-rsa'):
```

```
            return key
    raise KeyError("ssh host key not found")


def main():
    server = get_server(get_nova_creds(), "my-server")
    netname = "my-network"
    (fixed_ip, floating_ip) = server.networks[netname]
    # Remove existing key, if any
    remove_hostkey(floating_ip)
    output = server.get_console_output()
    key = get_hostkey_from_console(output)
    with open(os.path.expanduser("~/.ssh/known_hosts"), 'a') as f:
        f.write("{0} {1}\n".format(floating_ip, key))


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

OpenStack Image (glance)

The OpenStack Image Service (`glance`) is responsible for managing a catalog of VM images that the Compute Service uses.

Authenticating against the glance endpoint

The OpenStack Image (`glance`) Python API has some minor differences to the Compute API when doing the initial authentication. The `glance` API depends on information that it must obtain from the `keystone` API:

- The `glance` endpoint URL
- A `keystone` authentication token

Like the `nova` API, with the `glance` API, you can pass the API version as an argument or import the module directly. Listing 20 shows an example of how to authenticate against a `glance` endpoint with version 2 of the API.

Authenticating with the glance API

```
import keystoneclient.v2_0.client as ksclient
import glanceclient
creds = get_keystone_creds()
keystone = ksclient.Client(**creds)
glance_endpoint = keystone.service_catalog.url_for(service_type='image',
                                                   endpoint_type='publicURL')
glance = glanceclient.Client('2',glance_endpoint, token=keystone.auth_token)
```

Listing 21 shows an example in which the relevant `glance` module is imported directly.

Importing the glance module directly

```
import keystoneclient.v2_0.client as ksclient
import glanceclient.v2.client as glclient
creds = get_keystone_creds()
keystone = ksclient.Client(**creds)
glance_endpoint = keystone.service_catalog.url_for(service_type='image',
                                                   endpoint_type='publicURL')
glance = glclient.Client(glance_endpoint, token=keystone.auth_token)
```

List available images

Use the `Client.images.list` method to list the current images, as shown in Listing 22. Note that this method returns a generator, where the `list` methods in the `nova` API return list objects.

Retrieve a list of VM images

```
>>> import keystoneclient.v2_0.client as ksclient
>>> import glanceclient.v2.client as glclient
>>> creds = get_keystone_creds()
>>> keystone = ksclient.Client(**creds)
>>> glance_endpoint = keystone.service_catalog.url_for(service_type='image',
...                                                    endpoint_type='publicURL')
>>> glance = glclient.Client(glance_endpoint, token=keystone.auth_token)
>>> images = glance.images.list()
>>> images
<generator object list at 0x10c8efd70>
>>> images.next()
{u'status': u'active', u'tags': [], u'kernel_id':
u'8ab02091-21ea-434c-9b7b-9b4e2ae49591', u'container_format': u'ami', u'min_ram': 0,
u'ramdisk_id': u'd36267b5-7cae-4dec-b5bc-6d2de5c89c64', u'updated_at':
u'2013-05-28T00:44:21Z', u'visibility': u'public', u'file':
u'/v2/images/cac50405-f4d4-4715-b1f6-7f00ff5030e6/file', u'min_disk': 0,
u'id': u'cac50405-f4d4-4715-b1f6-7f00ff5030e6', u'size': 25165824, u'name':
u'cirros-0.3.1-x86_64-uec', u'checksum': u'f8a2eeee2dc65b3d9b6e63678955bd83',
u'created_at': u'2013-05-28T00:44:21Z', u'disk_format': u'ami', u'protected':
False, u'schema': u'/v2/schemas/image'}
```

Uploading an image to glance

Listing 23 shows an example of how to use the `glance` API to upload a file. You need to use version 1 of the API to create an image, because the Python API bindings have not implemented the `create` method for version 2.

Uploading an image to glance

```
import keystoneclient.v2_0.client as ksclient
import glanceclient
creds = get_keystone_creds()
keystone = ksclient.Client(**creds)
glance_endpoint = keystone.service_catalog.url_for(service_type='image',
                                                   endpoint_type='publicURL')
glance = glanceclient.Client('1',glance_endpoint, token=keystone.auth_token)
with open('/tmp/cirros-0.3.0-x86_64-disk.img') as fimage:
    glance.images.create(name="cirros", is_public=True, disk_format="qcow2",
                         container_format="bare", data=fimage)
```

Next steps

This article provides only a brief overview of the functionality the OpenStack Python APIs expose. Here are a few ways you can learn more about how the APIs work.

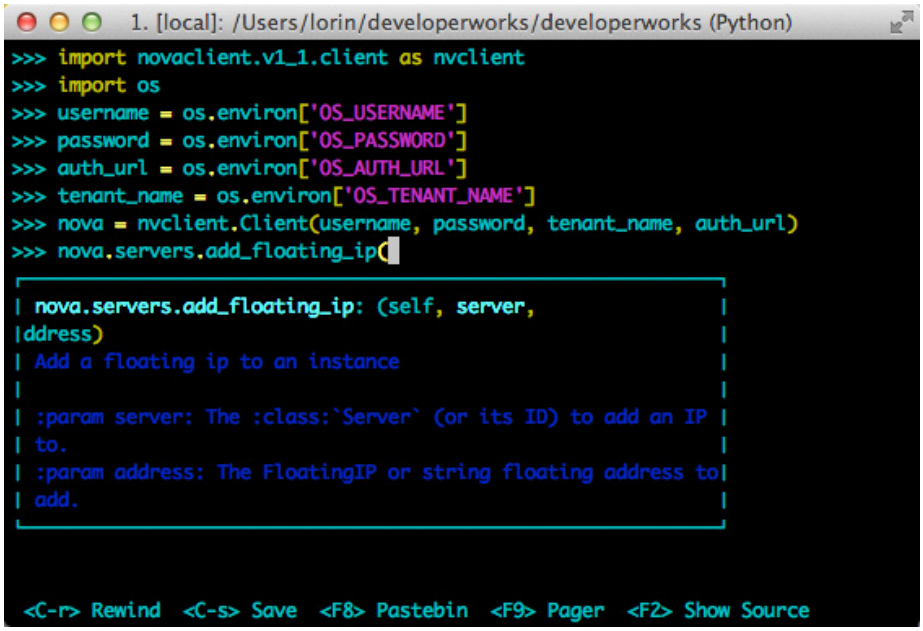
The official API documentation

The OpenStack project maintains documentation for all of OpenStack Python APIs. All of the APIs have auto-generated documentation for each module, class, and method. Some of the APIs have usage examples in the docs, and others don't.

Introspect the API

One of the best ways to learn about the APIs is to use them inside an interactive command-line Python interpreter. The bpython interpreter as an enhanced Python interpreter that displays valid method names as you type and automatically shows you the docstring of a function (see Figure 1).

The bpython automatic help display



Examine the CLI source code

One of Python's strengths is its readability, and there's no better way to learn the API than to read the source code. The packages are all hosted on github in the openstack group. For example, to get a copy of the `nova` API source code, run the following command:

```
git clone http://github.com/openstack/python-novaclient
```

Because the command-line clients are implemented using the API, each package conveniently ships with an example application.

For the `novaclient` API, the files of most interest are those in the novaclient/v1_1 directory, which contains the Python classes that form the API. The command-line commands on the shell are implemented as `do_*` methods in novaclient/v1_1/shell.py. For example, `nova flavor-list` is implemented as the `do_flavor_list` method, which ultimately calls the `Client.flavors.list` API method.

Examine other apps that use the Python APIs

Several other applications use the OpenStack Python APIs. The OpenStack Dashboard communicates with the various OpenStack services entirely using the Python APIs. It's a great example of an application that uses the Python APIs. In particular, look at the openstack_dashboard/api directory to see how the dashboard uses the Python API.

The OpenStack Client is an effort to unify the functionality across the existing clients into a single command-line interface. It uses the Python APIs from all of the other projects.

The Heat project is an orchestration layer designed to work with OpenStack and uses the APIs. In particular, look at the heat/engine/clients.py file.

Ansible is a Python-based configuration management tool that has several OpenStack modules that use the Python APIs. In particular, look at the library/cloud directory, which contains the Ansible OpenStack modules.

Once you learn how the Python APIs work, it's difficult to imagine going back to using the REST API or the command-line tools to build your OpenStack automation scripts.

Related topics

- OpenStack API Bindings
- Software Development Kits
- TryStack.org
- OpenStack Documentation