NUST School of Electrical Engineering and Computer Science
KTH Applied Information Security Lab

DOCUMENTATION ON ADDING ENCRYPTION TO OPENSTACK SWIFT

BY MUHAMMAD KAZIM

& MOHAMMAD RAFAY ALEEM

30/11/2013

TABLE OF CONTENTS

CHAPTER 1: Introduction to Swift

Swift also called the OpenStack object storage is a scalable object storage system. It is a multi-tenant, highly scalable and durable object storage system that was designed to store large amounts of unstructured data at low cost via a RESTful http API. Swift's usage ranges from small deployments for "just" storing VM images, to mission critical storage clusters for high-volume websites, to mobile application development, data analytics and private storage infrastructure-as-a-service.

The components that enable Swift to deliver high availability, high durability and high concurrency are:

Proxy Servers: Handles all incoming API requests.

Rings: Maps logical names of data to locations on particular disks.

Zones: Each Zone isolates data from other Zones. A failure in one Zone doesn't impact the rest of the cluster because data is replicated across the Zones.

Accounts & Containers: Each Account and Container are individual databases that are distributed across the cluster. An Account database contains the list of Containers in that Account. A Container database contains the list of Objects in that Container.

Objects: The actual data itself on which operation is to be performed.

Partitions: A Partition stores Objects, Account databases and Container databases. It's an intermediate 'bucket' that helps manage locations where data lives in the cluster.

More details about the swift architecture can be found here:

http://swiftstack.com/openstack-swift/architecture/.

CHAPTER 2: Deploying OpenStack

In order to perform development in OpenStack, they have provided a developers version of OpenStack called as DevStack. It is easy to deploy and provides a development platform for OpenStack.

DevStack is a documented shell script to build complete OpenStack development environments and is located at http://devstack.org/.

The recommended way to install DevStack is on Linux running on a virtual machine. Typical configuration used for installing Devstack is as follows:

- Ubuntu 12.04 LTS as host OS.
- Ubuntu 12.04 LTS as guest OS running on VMware Player.

For deploying DevStack on your virtual machine, you have to make sure that git is installed inside your VM.

Install it using the following commands:

```
apt-get update
apt-get install git
```

Clone the DevStack repo using the following:

```
git clone https://github.com/openstack-dev/devstack.git
```

Now run the following commands:

```
cd devstack
```

We are not going to execute the stack.sh script right now because we have to modify our localrc to disable all other services except swift and keystone. The localrc file (located in devstack) contains some of the configuration primitives which can be modified as-per-need basis. At this point, create a localrc file in devstack directory and add following lines in it:

```
ADMIN_PASSWORD=password
MYSQL_PASSWORD=password
RABBIT_PASSWORD=password
```

SERVICE_PASSWORD=password
SERVICE_TOKEN=tokentoken

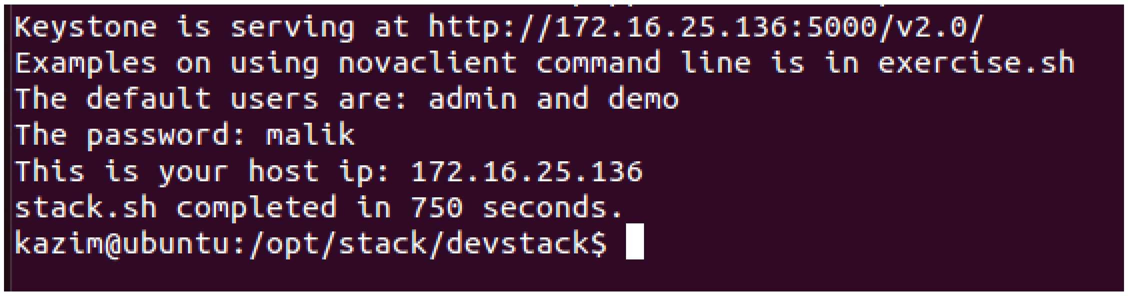
Now add the following lines:

disable_all_services
enable_service key mysql s-proxy s-object s-container s-account

We are now ready to start OpenStack. cd into the devstack directory and execute the stack.sh script using the following command.

```
./stack.sh
```

After the execution is complete, you will see output similar to one shown in Figure 1:



```
Keystone is serving at http://172.16.25.136:5000/v2.0/
Examples on using novaclient command line is in exercise.sh
The default users are: admin and demo
The password: malik
This is your host ip: 172.16.25.136
stack.sh completed in 750 seconds.
kazim@ubuntu:/opt/stack/devstack$
```

Figure 1: Output of the running DevStack

You have a working deployment of OpenStack, configured to run only Keystone and Swift. Hence don't expect the OpenStack GUI (Horizon) to work right now as it is not enabled.

Adding objects to OpenStack Swift:

At this point, you should already be familiar with Keystone, its relation with every other component of OpenStack and its necessity. Keystone is an OpenStack project that provides Identity, Token, Catalog and Policy services for use specifically by projects in the OpenStack family. The Keystone Identity Service allows clients to obtain tokens that can be used to access OpenStack cloud services.

To add objects to Swift, go into the DevStack directory. The absolute path of DevStack Swift is `/opt/stack/devstack`. Enter the following command in your terminal:

```
source openrc
```

This command authorizes you for using OpenStack Swift and generates a token for you. Without it, you cannot access Swift. Now that we are authorized, enter the following command:

```
swift stat
```

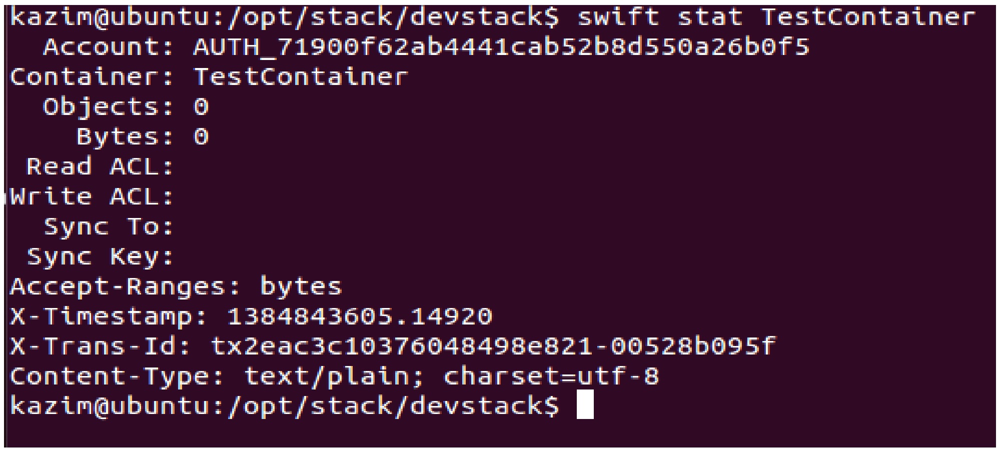
You should see something like this:

```
Account: AUTH_54805eb6837e40818998e59ee6ea40c7
Containers: 0
Objects: 0
Bytes: 0
Accept-Ranges: bytes
X-Timestamp: 1379268007.20819
X-Trans-Id: tx93f58db540574b228fd87effd76c5221
Content-Type: text/plain; charset=utf-8
```

Now that we have got complete access to Swift, let's create containers and objects. Enter the following command to create a container in OpenStack Swift:

```
swift post TestContainer
```

This will create a new container called "TestContainer". Status of the new container can be seen be giving the command "Swift stat TestContainer". Containers are essentially databases that are used to store information about objects. Consider them as directories where you can store files (objects). Figure 2 shows the status of new container created.



```
kazim@ubuntu:/opt/stack/devstack$ swift stat TestContainer
  Account: AUTH_71900f62ab4441cab52b8d550a26b0f5
Container: TestContainer
  Objects: 0
    Bytes: 0
 Read ACL:
Write ACL:
  Sync To:
 Sync Key:
Accept-Ranges: bytes
X-Timestamp: 1384843605.14920
X-Trans-Id: tx2eac3c10376048498e821-00528b095f
Content-Type: text/plain; charset=utf-8
kazim@ubuntu:/opt/stack/devstack$
```

Figure 2: Status of Container named TestContainer

Now enter “swift stat”. You will see that your account now contains a single container. You can also physically see a container! Open your root directory and navigate to /opt/stack/data/swift/1/sdbX/Y/sdbZ/containers/.

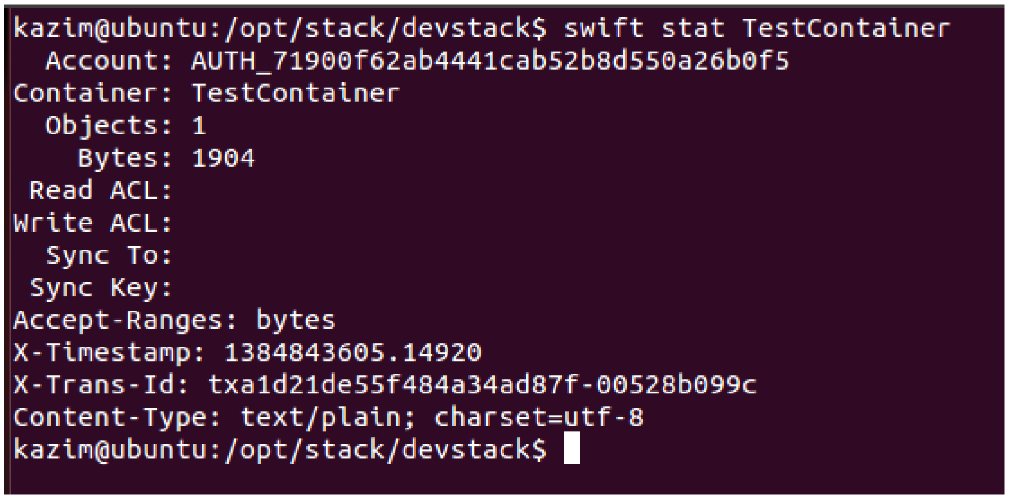
Go deep into the nested directories inside the container directory and you will eventually find a file named as xxx.db. This is the database that will hold information about objects stored by Swift.

If you want to know more about the container, enter “`swift stat TestContainer`”.
Now let’s upload a test file. Navigate to any directory containing a desired file to be uploaded and enter the following command:

```
swift upload TestContainer Test_File
```

where Test_File is a file in devstack directory.

Enter “`swift stat TestContainer`” to view your container properties.



```
kazim@ubuntu:/opt/stack/devstack$ swift stat TestContainer
  Account: AUTH_71900f62ab4441cab52b8d550a26b0f5
Container: TestContainer
  Objects: 1
    Bytes: 1904
 Read ACL:
Write ACL:
  Sync To:
 Sync Key:
Accept-Ranges: bytes
X-Timestamp: 1384843605.14920
X-Trans-Id: txa1d21de55f484a34ad87f-00528b099c
Content-Type: text/plain; charset=utf-8
kazim@ubuntu:/opt/stack/devstack$
```

Figure 3: Status of TestContainer after adding an object

That was it for some of the basic operations on Swift. You can view CLI help by issuing

“`swift -h`”.

CHAPTER 3: Debugging OpenStack

After DevStack is configured, navigate to opt/stack directory on your machine. All of the code for OpenStack components is located in their respective directories. Now navigate to swift/swift directory. Following are the most important directories:

- account
- common
- container
- obj
- proxy

These directories contain code for their respective components. For instance, account will contain code for the account server, obj will contain the code for object server and so on.

We will debug the code in a bit but let's, first make some important things clear.
All the code in these directories are downloaded from online Swift GitHub repos. You can modify these repos to download and replace your code instead of the default ones. This is very important. As soon as you start working on your own code, you will have to add it to some version control to keep track of your changes. You can then share that repository with anyone who can then download a working copy such that his OpenStack deployment will use your code instead of the default one.

Repository locations are stored in stackrc file in the devstack directory. Let's now replace the default location of the swift repo with your own. Go to https://github.com/openstack/swift
and fork the repo. After that, copy the forked repo's clone URL and paste it in stackrc `SWIFT_REPO` (replacing the previous one).
Undeploy and deploy OpenStack by executing "`./unstack.sh`" and "`./stack.sh`" respectively.

This deployment will now run your code (in this case Swift code).

Now we can start debugging OpenStack Swift code.

At this point, you should be comfortable with the *Python Debugger (pdb)* because this is the tool we would rely on for debugging Swift. We have to do some minor tweaks so let's do them first.

Locate utils.py in common directory of swift and add “stdio_files = []”after the line that reads “stdio_files = [sys.stdin, sys.stdout, sys.stderr]”. So now your utils.py looks like:

```
stdio_files = [sys.stdin, sys.stdout, sys.stderr].
stdio_files = []
```

Now let’s set our first breakpoint. Navigate to the obj directory and open server.py.
Navigate to the class ObjectController's PUT function and add the following lines:

```
import pdb
pdb.set_trace()
```

This sets a breakpoint in the PUT function of the ObjectController class and will hit each time you try uploading a file to the server. Save the file and exit.
The next step is the actual debugging.

Ensure that you have two terminals opened side by side. Enter screen -r in one of the terminal (call it Termial2). Hit Ctrl + A + X where X corresponds to the number against s-object.

Now on the other terminal (Terminal1), issue upload commands to a swift container of your choice.

You will be able to hit breakpoint onTerminal2 where you can debug it using normal pdb commands. In case if something goes wrong in Terminal2, hit Ctrl + C, press the up arrow key and hit enter. This will restart the object server and you are good to go again with your debugging.

CHAPTER 4: OpenStack Object Server Code Breakdown & Object Encryption

In this chapter, we are going to see how Swift Object Server works. Later on, we will create and add encryption to this Object Server and test it.

Navigate to the opt/stack directory on your virtual machine.
Now, navigate to swift/swift/obj directory and open server.py in your favorite Python editor. server.py contains most of the Swift Object Server code.

There are only two classes in server.py.
– DiskFile class for file I/O
– ObjectController class which handles all the requests made by the user to the Swift object storage.

Since OpenStack Swift API is implemented as a set of ReSTful web services, you will find the conventional requests methods such as PUT, POST and GET inside the ObjectController class.

The point of our focus here is file PUT so we are going to skip the breakdown of POST and GET functions.

To upload a file to the object storage, a put request is issued and handled by PUT function of the ObjectController class.

At this point, it would be a good idea to setup breakpoints in the ObjectController PUT function and upload a file a file to start debugging.

When you upload a file to the object storage, it is not written in a single go. Each file is read into a number of small chunks and are stored in a temporary file which acts as a buffer. Contents from the temporary file are then written to a permanent file when buffer size reaches a certain threshold value (defualt is 512 MB).

Following are some of the important lines/loops from the PUT function.

```
- file = DiskFile(self.devices, device, partition, account, 
container,
            obj, self.logger,
      disk_chunk_size=self.disk_chunk_size,
            origin_disk_chunk_size=self.origin_disk_chunk_size)
```

```
- with file.mkstemp() as fd:

- for chunk in iter(lambda: reader(self.network_chunk_size),
''):

- while chunk:
          written = os.write(fd, chunk)
          chunk = chunk[written:]
          # For large files sync every 512MB (by default)
     written
          if upload_size - last_sync >= self.bytes_per_sync:
          tpool.execute(fdatasync, fd)
          drop_buffer_cache(fd, last_sync, upload_size -
     last_sync)
          last_sync = upload_size
```

With this basic understanding of how a call to PUT works, we will now talk about object encryption.
Since each of the chunks is written to a buffer, it would be wise to encrypt them before they are written to it. In this way, contents are copied from the buffer to the actual file will be encrypted such that when all chunks are written to the actual file, the file will be encrypted.

To handle encryption, we have created a separate Encryptor.py file that will handle the encryption and return encrypted chunks. Our Encryption classes use the popular "M2Crypto" package to perform encryption. M2Crypto is present as a as a fully supported package in the Ubuntu 10.04 and later repositories. It can be installed using the command,

```
sudo apt-get install python-m2crypto.
```

The following line in the PUT function of server.py will perform and return an encrypted chunk.

```
enc = encrypted();
enc.EncryptFile(fname)
```

Parameter fname is the path of the file that is to be written to disk.
Writing the encrypted chunks would be similar to our previous descriptions.
Note that the current implementation uses a constant key that is saved in a configuration file. Our implementation lacks a functional keystore. See the following code used for encryption in the '`encrypted`' class in Encrypt.py file.

```
class encrypted:

     def __init__(self):

          self.key=128
          self.alg='aes_128_cbc'
          self.key="123452345"
          self.iv = '3141527182810345'

     def set_state(self,  op=ENC):

           self.cipher =M2Crypto.EVP.Cipher(alg=self.alg,
          key=self.key, iv=self.iv, op=op)

     def encrypt(self,msg):
          v = self.cipher.update(msg)
          v = v + self.cipher.final()
          return v
          #print 'Successful encryption'

     def EncryptFile(self,dname):
          os.chdir(dname)
          for filename in os.listdir('.'):
               #print "Opening File %s for Encryption"%filename
               with open(filename) as f:
                    content = f.read()
                    f.close()
          self.set_state(1)
          print "Content length is===",len(content)
          #while (len(content) % 16 != 0):
          #    content = content + ' '
          content = self.encrypt(content)
          print 'Successful Encryption'

          faname = os.path.join(dname, filename)
          f = open(faname, 'w')
          f.write(content)
          f.close();
```

Decryption is analogous to our description above but will occur in the ObjectController class. When a request is made to retrieve a file, the object server returns an iterator to retrieve that file. It is the responsibility of that GET function of server.py to decrypt file before returning it.

Following code in Encrypt.py performs the decryption.

```
def DecryptFile(self,dname,obj):
          objectname = obj
          os.chdir(dname)
```

```
	#import pdb
	#pdb.set_trace()
	for filename in os.listdir('.'):
		with open(filename) as f:
			content = f.read()
			f.close()

	self.set_state(0)
	content = self.encrypt(content)
	print 'Successful Decryption'

	#completeName = os.path.abspath("~/fname" %
#objectname)
	completeName = os.path.join(dname, filename)
	f = open(completeName, 'w')
	f.write(content)
	f.close();
```

CHAPTER 5: Deployment and Testing

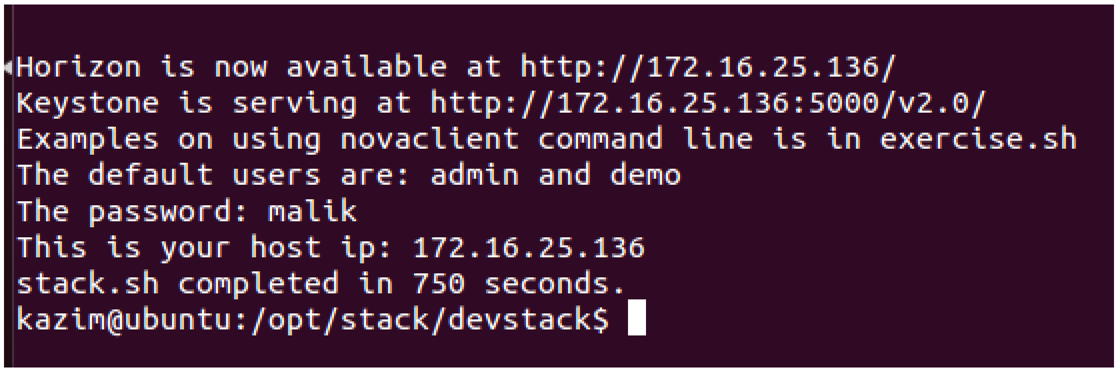
Deployment:

For deploying the Swift encryption code in OpenStack, following steps need to be followed:

1. It is recommend a fresh VM of Ubuntu desktop, as previously saved changes in Devstack can get lost.
2. Create a folder in /opt named stack.
3. Go to /opt/stack and copy the devstack.tar folder provided with code.
4. From command line, extract the devstack.tar.gz folder in /opt/stack using the following command: `tar xpvzf devstack.tar.gz devstack/`
5. Now extract the swift.tar.gz with encryption code, using same command in the /opt/stack folder.
6. Run the Devstack script and it will download nova, keystone, glance, horizon and others services from internet.
7. Change glance/etc/glance-api.conf file to use Swift store (as mentioned in chapter 6).
8. Run the script ./stack.sh.
9. After successful completion of script, goto /opt/stack/data/swift/1/sdb1/objects. Delete the existing objects here so that you can check for the objects you add by yourself.
10. From dashboard, add an .img Qcow2 bootable image. Check the image you just uploaded from dashboard in /opt/stack/data/swift/1/sdb1/objects. This image will be stored encrypted. You can verify it using image size and md5sum as mentioned in documentation.
11. Launch an instance from this image, the image will be decrypted before use by VM and VM should be launched in 'active' state. You should be able to access its console.

Testing:

Complete OpenStack deployment through DevStack can be used accessed through dashboard. The URL on which Horizon is running after the successful execution of DevStack can be entered to browser. Login using the credentials (username and password) obtained from Keystone after successful execution of DevStack as shown in figure 4.

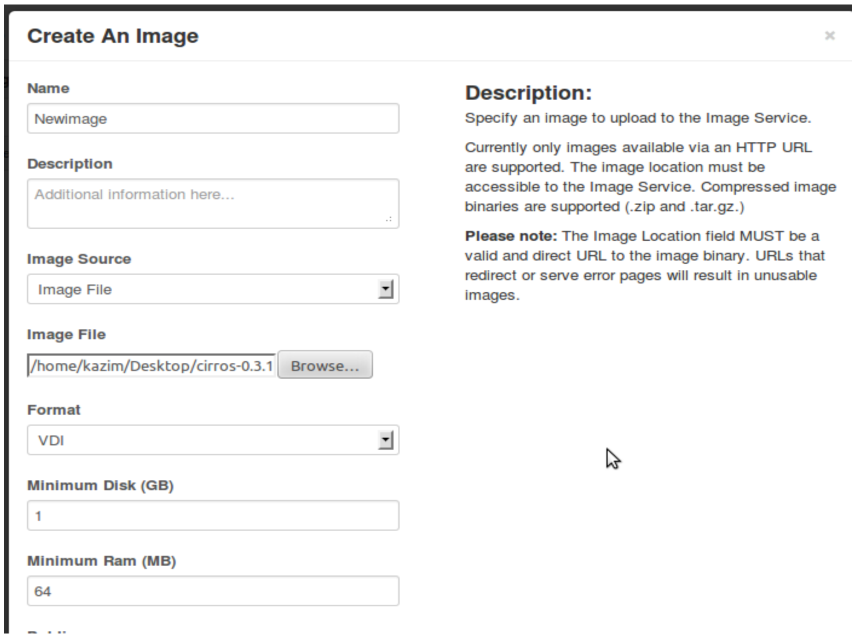


```
Horizon is now available at http://172.16.25.136/
Keystone is serving at http://172.16.25.136:5000/v2.0/
Examples on using novaclient command line is in exercise.sh
The default users are: admin and demo
The password: malik
This is your host ip: 172.16.25.136
stack.sh completed in 750 seconds.
kazim@ubuntu:/opt/stack/devstack$
```

Figure 4: Horizon URL and User Credentials

After the login, new and custom images can be added to OpenStack. The custom bootable image for OpenStack named cirros can be downloaded from https://launchpad.net/cirros/+download. For test case, we can download a Qcow2 bootable image named “cirros-0.3.0-i386-disk.img” from above link.

After downloading the image, go to “Images” tab on left side of Dashboard. Click on “create image”, you can browse to the qcow image and select it to upload. You can name the image and select its format. All properties during creation of custom image named “Newimage” are shown in figure 5.


Create An Image

Name
Newimage

Description
Additional information here...

Image Source
Image File

Image File
/home/kazim/Desktop/cirros-0.3.1

Format
VDI

Minimum Disk (GB)
1

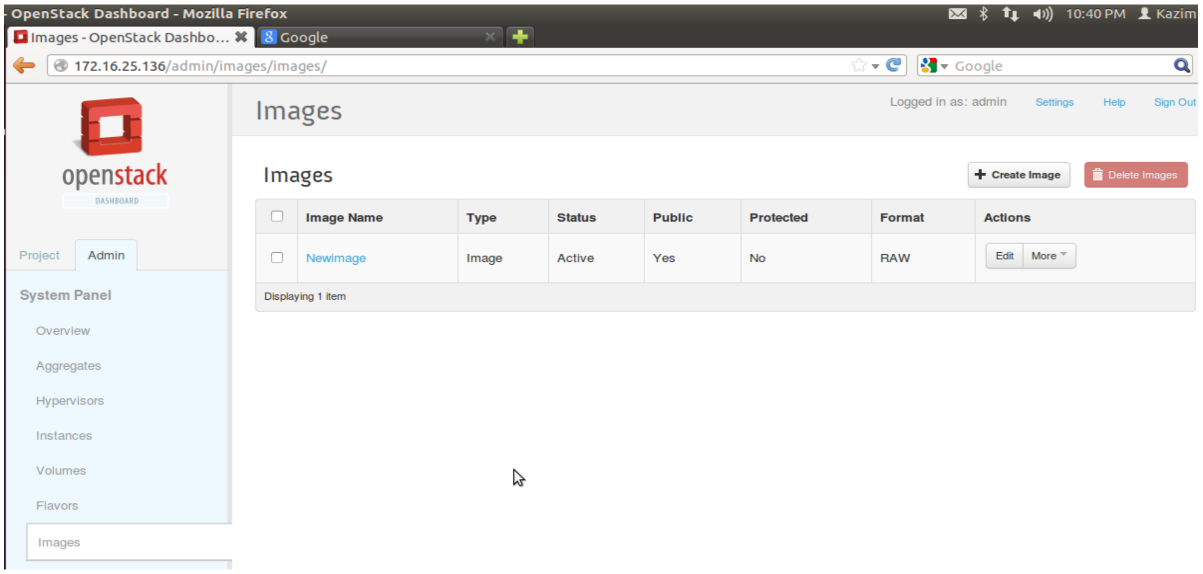
Minimum Ram (MB)
64

Description:
Specify an image to upload to the Image Service.

Currently only images available via an HTTP URL are supported. The image location must be accessible to the Image Service. Compressed image binaries are supported (.zip and .tar.gz.)

Please note: The Image Location field MUST be a valid and direct URL to the image binary. URLs that redirect or serve error pages will result in unusable images.

Figure 5: Image creation process in OpenStack



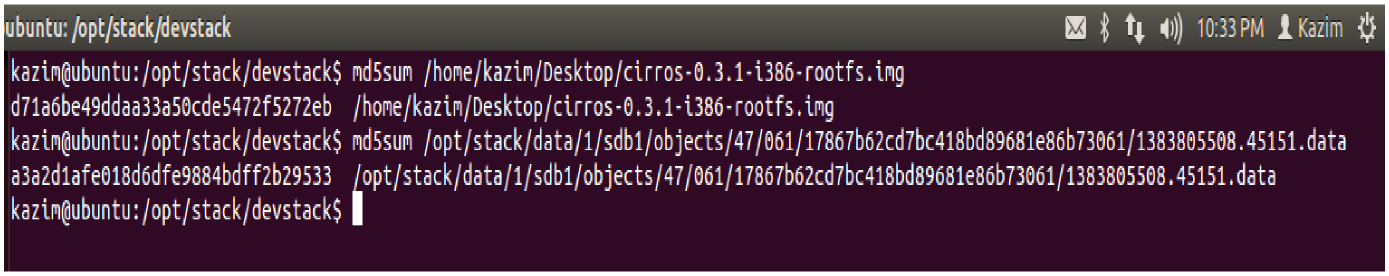
Images

Image Name	Type	Status	Public	Protected	Format	Actions
Newimage	Image	Active	Yes	No	RAW	Edit More

Displaying 1 item

Figure 6: Successful creation of image on OpenStack

This image is present in encrypted state on the disk. The location of image can be seen in the directory /opt/stack/data/swift/1/sdbX/Y/sdbZ/Objects/. To verify that the image is encrypted, we can see the size of image uploaded and the original image that was downloaded. Moreover, we can use Md5sum to take hashes of the two files and compare them. Figure 7 shows the hash of original .img file before encryption and after encryption.

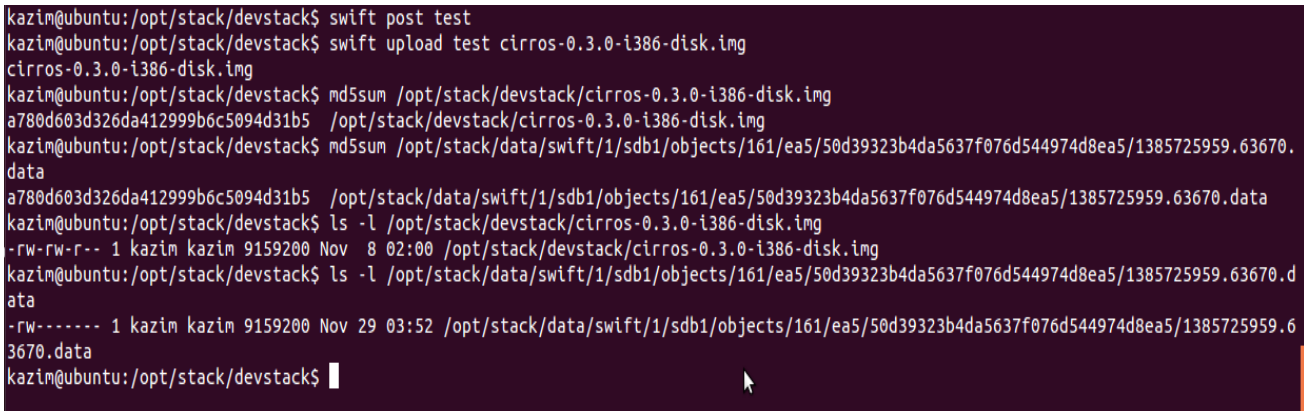


```
ubuntu: /opt/stack/devstack
kazim@ubuntu:/opt/stack/devstack$ md5sum /home/kazim/Desktop/cirros-0.3.1-i386-rootfs.img
d71a6be49ddaa33a50cde5472f5272eb  /home/kazim/Desktop/cirros-0.3.1-i386-rootfs.img
kazim@ubuntu:/opt/stack/devstack$ md5sum /opt/stack/data/1/sdb1/objects/47/061/17867b62cd7bc418bd89681e86b73061/1383805508.45151.data
a3a2d1afe018d6dfe9884bdff2b29533  /opt/stack/data/1/sdb1/objects/47/061/17867b62cd7bc418bd89681e86b73061/1383805508.45151.data
kazim@ubuntu:/opt/stack/devstack$
```

Figure 7: Md5sum of original file and encrypted files

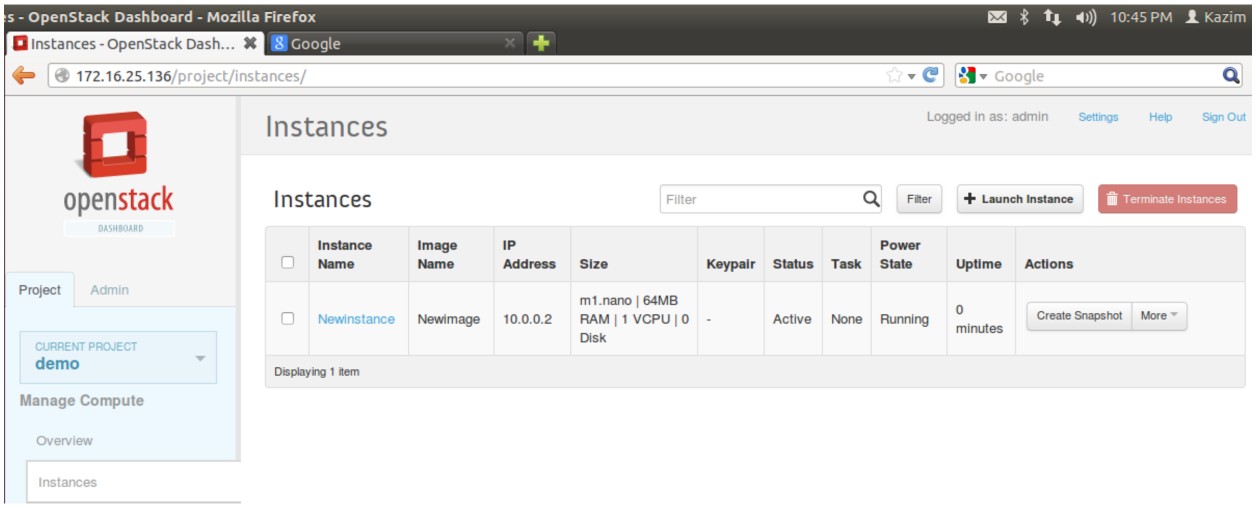
Figure 8 shows the md5sum and size of images added in Swift without encryption. Without encryption, the md5sum and size of original and uploaded images are same.

To launch an instance using the encrypted image, click on “Instance” tab on left and then on “launch instance”. Select the custom image named “Newimage” to be used by the instance. The instance created using the “Newimage” is shown in figure 9.



```
kazim@ubuntu:/opt/stack/devstack$ swift post test
kazim@ubuntu:/opt/stack/devstack$ swift upload test cirros-0.3.0-i386-disk.img
cirros-0.3.0-i386-disk.img
kazim@ubuntu:/opt/stack/devstack$ md5sum /opt/stack/devstack/cirros-0.3.0-i386-disk.img
a780d603d326da412999b6c5094d31b5  /opt/stack/devstack/cirros-0.3.0-i386-disk.img
kazim@ubuntu:/opt/stack/devstack$ md5sum /opt/stack/data/swift/1/sdb1/objects/161/ea5/50d39323b4da5637f076d544974d8ea5/1385725959.63670.
data
a780d603d326da412999b6c5094d31b5  /opt/stack/data/swift/1/sdb1/objects/161/ea5/50d39323b4da5637f076d544974d8ea5/1385725959.63670.data
kazim@ubuntu:/opt/stack/devstack$ ls -l /opt/stack/devstack/cirros-0.3.0-i386-disk.img
-rw-rw-r-- 1 kazim kazim 9159200 Nov  8 02:00 /opt/stack/devstack/cirros-0.3.0-i386-disk.img
kazim@ubuntu:/opt/stack/devstack$ ls -l /opt/stack/data/swift/1/sdb1/objects/161/ea5/50d39323b4da5637f076d544974d8ea5/1385725959.63670.d
ata
-rw------- 1 kazim kazim 9159200 Nov 29 03:52 /opt/stack/data/swift/1/sdb1/objects/161/ea5/50d39323b4da5637f076d544974d8ea5/1385725959.6
3670.data
kazim@ubuntu:/opt/stack/devstack$
```

Figure 8: Md5sum and size of images without Swift encryption



Instances

Instances

	Instance Name	Image Name	IP Address	Size	Keypair	Status	Task	Power State	Uptime	Actions
	Newinstance	Newimage	10.0.0.2	m1.nano | 64MB RAM | 1 VCPU | 0 Disk	-	Active	None	Running	0 minutes	Create Snapshot More

Displaying 1 item

Figure 9: Custom instance successfully running

The image on disk will be automatically decrypted when you click on "launch". The instance can be used by the user and after the instance termination; the image will be saved in encrypted form on disk. Since the image is present is encrypted state on disk, the data on image cannot be read and accessed unless it is decrypted. Hence, the confidentiality of image in Cloud storage is preserved.

CHAPTER 6: Common errors and Notes

Configuring Glance to store virtual machine images:

The configuration file for the Image API is /etc/glance/glance-api.conf. To configure glance for using Swift, modify the value of parameter '`default_store`' in the `/glance/etc/glance-api.conf` file.

```
default_store = swift
```

Enable and Disable any service in OpenStack:

The devstack directory contains a file named localrc. localrc file can be used to enable or disable or any service of OpenStack. To use Swift for development, only Keystone and Mysql can be enabled with Swift after disabling all other services. It can be done using the following commands:

```
disable_all_services

enable_service key mysql s-proxy s-object s-account s-container
```

Similarly, any other OpenStack service can be enabled by adding its name next to enable_service. By default Devstack downloads all the services.

Using Devstack with proxy server:

For running Devstack behind the proxy server, following lines need to added to the localrc file in /opt/stack/devstack folder. It is assumed that proxy address is '10.1.11.11' with port '8080'.

```
http_proxy = http://10.1.11.11:8080/
https_proxy = http://10.1.11.11:8080/
export no_proxy = "localhost,127.0.0.1"

HOST_IP=localhost
SERVICE_HOST=$HOST_IP
IMAGE_HOST=$HOST_IP
IDENTITY_HOST=$HOST_IP
```

Maintaining Logs for Devstack:

To maintain logs for Devstack, following line needs to be added to the localrc file in /opt/stack/devstack directory. The logs will be maintained in /opt/stack/logs directory.

```
LOGFILE=$DEST/logs/stack.sh.log
```

Common Errors:

1. E: Could not get lock /var/lib/dpkg/lock - open (11 Resource temporarily unavailable):

A common error encountered while running a Devstack script is:

E: Could not get lock /var/lib/dpkg/lock - open (11 Resource temporarily unavailable.

To solve this issue, run the following commands in devstack directory before re-running script.

```
sudo rm /var/lib/apt/lists/lock

sudo rm /var/cache/apt/archives/lock
```

2. Invalid Nova Credentials (Unauthorized HTTP 401)

The admin user credentials also need to be put into /etc/nova/api-paste.ini. The values that need to be added in .ini file are admin_tenant_name, admin_user, and admin_password. More details can be found here, https://ask.openstack.org/en/question/3571/nova-client-error-unauthorized-http-401/.