Swift Installation(Ubuntu)

- The instructions in this presentation show a fairly generic installation for a small deployment.
- The exact values you use when you are installing will vary depending on your needs and the physical or virtual machine(s) you are using.

Jannatun Noor

Downloading OpenStack Swift

- The official releases of OpenStack Swift are available on GitHub
- **Dependencies**
- Depending on your distribution, there might be some dependencies that will need to be installed on your system before we install Swift.
- **On apt-based systems (Debian and derivatives):**
  apt-get install git curl gcc memcached rsync sqlite3 xfsprogs git-core libffi-dev python-setuptools

  apt-get install python-coverage python-dev python-nose python-simplejson python-xattr python-eventlet python-greenlet python-pastedeploy python-netifaces python-pip python-dnspython python-mock

- After you install the dependencies, you will install the Swift command-line interface and Swift.
- After installation, you will need to configure and initialize the Swift processes before your cluster and node(s) will be up and running

Installing the Swift CLI (python-swiftclient)

- Install the Swift command-line interface (CLI) from GitHub with the following commands:

```
cd /opt
git clone https://github.com/openstack/python-swiftclient.git
cd /opt/python-swiftclient;
sudo pip install -r requirements.txt;
python setup.py install;
cd ..
```

Installing Swift

- Installing Swift from GitHub is similar to installing the CLI; use the following commands:

```
cd /opt
git clone https://github.com/openstack/swift.git
cd /opt/swift ;
 sudo python setup.py install;
cd ..
```

Copying in Swift Configuration Files

- Create a directory for Swift and then copy in the sample configuration files so you can use them as a template during configuration:

```
mkdir -p /etc/swift
cd /opt/swift/etc
cp account-server.conf-sample /etc/swift/account-server.conf
cp container-server.conf-sample /etc/swift/container-server.conf
cp object-server.conf-sample /etc/swift/object-server.conf
cp proxy-server.conf-sample /etc/swift/proxy-server.conf
cp drive-audit.conf-sample /etc/swift/drive-audit.conf
cp swift.conf-sample /etc/swift/swift.conf
```

Copying in Swift Configuration Files(cont.)

- At this point, you should be able to run the swift-init command with the help flag and get the help message:
  swift-init -h

- This command is similar to service or start|stop ;it allows you to start/stop/reload Swift processes.
- However, don’t try to start Swift up just yet.

Configuring Swift

- Before starting the processes, the newly installed Swift needs configuring.
- You will need to create the builder files, add the drives, and build the rings.
- After that we will discuss how to set up logging, start services, and check our work.

Adding Drives to Swift

- Storing data on hard drives is the fundamental purpose of Swift, so adding and formatting the drives is an important first step.
- Swift replicates data across disks, nodes, zones, and regions, so there is no need for the data redundancy provided by a RAID controller.
- Parity-based RAID, such as RAID5, RAID6, and RAID10, harms the performance of a cluster and does not provide any additional protection because Swift is already providing data redundancy.

Adding Drives to Swift(cont.)

- **Finding drives**
- First, you need to identify what drives are in your system and determine which should be formatted and which are system drives that shouldn't be formatted.
- From the command line of the machine, run:

  df
- It will return something similar to:

```
Filesystem      1K-blocks       Used    Available       Use%    Mounted on
/dev/sda1       27689156        8068144 18214448        31%     /
none    1536448 672     1535776 1%      /dev
none    1543072 1552    1541520 1%      dev/shm

none    1543072 92      1542980 1%      /var/run
none    1543072 0       1543072 0%      /var/lock
```

- This should list all mounted filesystems. Just note which drive is used for booting so you don't accidentally format it later; very often it is the sda device mounted on the root / directory. But you can can verify that by poking around in /proc/mounts and finding which directory is the rootfs:

  cat /proc/mounts

Adding Drives to Swift(cont.)

- It will return something similar to:

```
rootfs / rootfs rw 0 0
none /sys sysfs rw,nosuid,nodev,noexec,relatime 0 0
none /proc proc rw,nosuid,nodev,noexec,relatime 0 0
none /dev devtmpfs
rw,relatime,size=1536448k,nr_inodes=211534,mode=755 0 0
none /dev/pts devpts
rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000 0 0
fusectl /sys/fs/fuse/connections fusectl rw,relatime 0 0
/dev/disk/by-uuid/c9402f9d-30ed-41f2-8255-d32bdb7fb7c2 / ext4 rw,relatime 0 0
none /sys/kernel/debug debugfs rw,relatime 0 0
none /sys/kernel/security securityfs rw,relatime 0 0
none /dev/shm tmpfs rw,nosuid,nodev,relatime 0 0
none /var/run tmpfs rw,nosuid,relatime,mode=755 0 0
none /var/lock tmpfs rw,nosuid,nodev,noexec,relatime 0 0
binfmt_misc /proc/sys/fs/binfmt_misc binfmt_misc
rw,nosuid,nodev,noexec,relatime 0 0
gvfs-fuse-daemon /home/joe/.gvfs fuse.gvfs-fuse-daemon
rw,nosuid,nodev,relatime,user_id=1000,group_id=1000 0 0
```

- The output tells you that the root filesystem is /, which you know from the df command is mounted on /dev/sda1. That is where the system will boot.

Adding Drives to Swift(cont.)

- To find out what other block devices the operating system knows about, look in the /sys/block directory. This should list all the block devices known to the system:

  ls /sys/block

- The command will return something similar to:

  loop0 loop3 loop6 ram1 ram12 ram15 ram3 ram6 sda sdd sdg sdj sdm
  loop1 loop4 loop7 ram10 ram13 ram19 ram4 ram7 sdb sde sdh sdk
  loop2 loop5 ram0 ram11 ram14 ram2 ram5 ram8 sdc sdf sdi sdl

- The devices you're interested in are the ones starting with sd in this output, but other systems might have different prefixes. Also review which filesystems are formatted by running the following, which you might need to run as root:

  blkid

- It will return something similar to:

  /dev/sda1: UUID="c9402f9d-30ed-41f2-8255-d32bdb7fb7c2" TYPE="ext4"
  /dev/sda5: UUID="b2c9d42b-e7ae-4987-8e12-8743ced6bd5e" TYPE="swap"

-
- The TYPE value will tell you which filesystem is on the device (e.g., xfs or ext4). Naturally, you should not use ones that have swap as their type.

Adding Drives to Swift(cont.)

- **Labeling drives**
- It's important to keep track of which devices need to be mounted to which system location. This can be done with mounting by labels in a separate upstart script created for Swift.
- We suggest not using the fstab file for this purpose. /etc/fstab is used to automatically mount filesystems at boot time.
- You should label each device so you can keep an inventory of each device in the system and properly mount the device.

  ```
  mkfs.xfs -f -L d1 /dev/sdb
  mkfs.xfs -f -L d2 /dev/sdc
  ```
- Swift uses extended attributes, and the XFS default inode size (currently 256) is recommended.
- Notice that we didn't create device partitions.
- Partitioning is an unnecessary abstraction for Swift because you're going to use the entire device.

Adding Drives to Swift(cont.)

- **Mounting drives**
- The next step is to tell the operating system to attach (mount) the new XFS filesystems somewhere on the devices so that Swift can find them and start putting data on them.
- Create a directory in /srv/node/ for each device as a place to mount the filesystem.
- We suggest a simple approach to naming the directories, such as using each device's label (d1, d2, etc):

  ```
  mkdir -p /srv/node/d1
  mkdir -p /srv/node/d2
  ```

- For each device (dev) you wish to mount, you will use the mount command:

  // _mount -t xfs -o noatime,nodiratime,logbufs=8 -L_ <dev label> <dev dir>

- Run the command for each device:

  ```
  mount -t xfs -o noatime,nodiratime,logbufs=8 -L d1 /srv/node/d1
  mount -t xfs -o noatime,nodiratime,logbufs=8 -L d2 /srv/node/d2
  ...
  ```

Adding Drives to Swift(cont.)

- **Swift User**
- After the drives are mounted, a user needs to be created with read/write permissions to the directories where the devices have been mounted.
- The default user that Swift uses is named swift .
- Different operating systems have different commands to create users, looking something like this:

  useradd swift
- Next, give the swift user ownership to the directories:

  chown -R swift:swift /srv/node

Adding Drives to Swift(cont.)

- **Creating scripts to mount the devices on boot**
- The mount commands we ran earlier take effect only during the current boot. Because we need them to run automatically on reboot, we'll put the commands in a script called mount_devices that contains the mount commands we just ran.
- This new script should be created in the /opt/swift/bin/ directory:

```
mount -t xfs -o noatime,nodiratime,logbufs=8 -L d1 /srv/node/d1
mount -t xfs -o noatime,nodiratime,logbufs=8 -L d2 /srv/node/d2
...
mount -t xfs -o noatime,nodiratime,logbufs=8 -L d36 /srv/node/d36
```

- Next, create an Upstart script in the /etc/init directory called start_swift.conf with the following commands:

```
description "mount swift drives"
start on runlevel [234]
stop on runlevel [0156]
exec /opt/swift/bin/mount_devices
```

Adding Drives to Swift(cont.)

- Be sure the script is executable:
  chmod +x /opt/swift/bin/mount_devices
- Test the scripts by rebooting the system:
  reboot
- When the system restarts, log on and make sure everything mounted properly with the configuration settings by running the mount command:
  mount
- You should see something similar to:
  /dev/sda1 on / type ext4 (rw,errors=remount-ro, commit=0)
  ...
  ...
  /dev/sdc on /srv/node/d1 type xfs
  (rw,noatime,nodiratime,logbufs=8)

Storage Policies

- Released with Swift 2.0, storage policies let Swift operators define space within a cluster that can be customized in numerous ways, including by location, replication, hardware, and partitions to meet specific data storage needs.
- To implement storage polices, you must create them in the cluster and then apply them to containers.
- Creating user-defined storage policies is a two-step process.
- The policies are first declared in the swift.conf file with name and index number.
- Then their index number is used to create the corresponding builder files and rings.
- It’s important to understand that in the absence of user-defined policies, Swift defaults to an internally referenced storage policy 0, which is applied to all containers.
- If you want to use a different name for storage policy 0, you can add it to the swift.conf file

Creating storage policies

- Each user-defined storage policy is declared with a policy index number and name. If a policy is deprecated or the default, there are additional fields that can be used. The general format for a storage policy is:

```
[storage-policy:N]
name =
default =
(optional) deprecated=
//Here's an example
[storage-policy:1]
name = level1
(optional) deprecated=yes
```

- The storage policy index, [storage-policy:N] , is required. The N is the storage policy index whose value is either 0 or a positive integer. It can’t be a number already in use, and once in use it can’t be changed.
- The policy name (name =) is required and can only be letters (case-insensitive), numbers, and dashes. It can’t be a name already in use, but the name can be changed.
- Before such a change, however, consider how it will affect systems using the current name.
- One storage policy must be designated as the default. When defining one or more storage policies, be sure to also declare a policy 0.

Creating storage policies

- When storage policies are no longer needed, they can be marked as deprecated with deprecated=yes rather than being deleted. A policy can’t be both the default and deprecated.
- When deprecated, a policy:
  - Will not appear in /info
  - Enables PUT / GET / DELETE / POST / HEAD requests to succeed on preexisting containers
  - Can’t be applied to a new container (a 400 Bad Request will be returned)

Creating storage policies

- Although storage policies are not necessary for this installation, we have included some to provide examples of how they are added and used.
- To declare a storage policy, edit the swift.conf file in the /etc/swift/ directory. The suggested placement in the file for storage polices is below the [swift-hash] section:

```
[swift-hash]
# random unique strings that can never change (DO NOT LOSE)
swift_hash_path_prefix = changeme
swift_hash_path_suffix = changeme
[storage-policy:0]
name = level1
[storage-policy:1]
name = level2
default = yes
[storage-policy:2]
name = level3
deprecated = yes
```

- Once the storage policies are declared, they can be used to create builder files.
- For each storage policy (storage-policy:N), an object-N.builder file will be created.

Creating the Ring Builder Files

- With the devices ready and storage polices created, it is time to create the builder files.
- You need to create a minimum of three files: account.builder, container.builder, and object.builder.
- You will also create a builder file for each user-defined storage policy that you are implementing in your cluster.

The create command

- Think of a builder file as a big database. It contains a record of all the storage devices in the cluster and the values the ring-builder utility will use to create a ring.
- The help menu for swift-ring-builder shows the format for the create command as:

  swift-ring-builder (account|container|object|object-n).builder create <part_power> <replicas> <min_part_hours>
- So the commands we will be running are:

  Swift-ring-builder account.builder create <part_power> <replicas> <min_part_hours>

  Swift-ring-builder container.builder create <part_power> <replicas> <min_part_hours>

  Swift-ring-builder object.builder create <part_power> <replicas> <min_part_hours>

  Swift-ring-builder object-n.builder create <part_power> <replicas> <min_part_hours>

The create command

- The three parameters the create command takes are:
  - part_power
    - Determines the number of partitions created in the storage cluster
    - total partitions in cluster = 2 ^ partitionpower
    - Let's look at an example for a large deployment of 1,800 drives that will grow to a total of 18,000 drives. If we set the maximum number of disks to 18,000, we can calculate a partition power that would allow for 100 partitions per disks:
    - partition power = log 2 (100 × 18, 000) ≈ 20.77
  - replicas
    - Specifies how many replicas you would like stored in the cluster
    - The replica count affects the system's durability, availability, and the amount of disk space used
    - It is set as part of the ring-building process to any real number, most often 3.0.
    - In the less-common instance where a non-integer replica count is used, a percentage of the partitions will have one additional replica; for instance, a 3.15 replica count means that 15% of the drives will have one extra replica for a total of 4.
    - The main reason for using a fractional replica count would be to make a gradual shift from one integer replica count to another, allowing Swift time to copy the extra data without saturating the network.
  - min_part_hours
    - Specifies the frequency at which a replica is allowed to be moved
    - A good default setting is 24 hours.

Running the create command

- For our storage policies, these values may be different. For this example, we will have storage policy 1 provide increased durability by setting its replica count to four. So we would use a part_power value of 17, replicas value of 4, and min_part_hours value of 1.

  ```
  cd /etc/swift
  Swift-ring-builder account.builder create 17 3 1
  Swift-ring-builder container.builder create 17 3 1
  Swift-ring-builder object.builder create 17 3 1
  ```

- You'll now see account.builder, container.builder, object.builder, object-1.builder, and object-2.builder in the directory.
- There should also be a backup directory, appropriately named backups.

Running the create command

- To take some mystery out of what was just created, let’s open an interactive Python session and show what is in an object.builder file. The contents are a Python data structure that you can view using a pickle file. Some of the fields will be familiar.

```
$ python
>>>
>>> import pickle
>>> print pickle.load(open('object.builder'))
{
    '_replica2part2dev': None,
    '_last_part_gather_start': 0,
    'in_part_hours': 24,
    'replicas': 3.0,
    'parts': 262144,
    'part_power': 18,
    'devs': [],
    'devs_changed': False,
    'version': 0,
    '_last_part_moves_epoch': None,
    '_last_part_moves': None,
    '_remove_devs': []
}
```

Adding Devices to the Builder Files

- Our next goal is to add the drives, their logical groupings (region, zone), and weight.
- This part of the configuration serves two purposes:
  - It marks failure boundaries for the cluster by placing each node into a region and zone.
  - It describes how much data Swift should put on each device in the cluster by giving it a weight.
- With the builder files created, we can now add devices to them. For each node, thestorage device in /srv/node will need an entry added to the builder file(s).
- Entries for devices are added to each file with this format:
  - Swift-ring-builder add account.builder <region><zone>-<IP>:6002/<device> <weight>

Adding Devices to the Builder Files

- IP
- You need to specify the IP address of the node for each device. This should be on the network segment designated for internal communication that will allow the nodes in the cluster to communicate with each other via rsync and HTTP.
- Port
- Each of the account, container, and object server processes can run on different ports.
- By default:
  - The account server process runs on port 6002.
  - The container server process runs on port 6001.
  - The object server process runs on port 6000.
- Weight
  - When starting out each drive is the same size, say 2 TB, you could give each drive a weight of 100.0, so that they all have the same likelihood of receiving a replica of a partition. Then, when it's time to add capacity and 3 TB drives are available, the new 3 TB drives would receive a weight of 150.0 because they are 50% larger.
  - With this weight, each 3 TB drive would recieve 1.5 times as many partition replicas as one of the 2 TB drives.
  - Another, often simpler, strategy is to use the drive's capacity as a multiplier. For example, the weight of a 2 TB drive would be 200.0 and a 3 TB drive would be 300.0.

Adding Drives

- Now that we've covered the parameters you need to specify when adding devices, we'll go ahead and add our devices to the builder files.

- The following adds the first device (d1) to the builder files:
  Swift-ring-builder account.builder add r1z1-127.0.0.1:6002/d1 100
  Swift-ring-builder container.builder add r1z1-127.0.0.1:6001/d1 100
  Swift-ring-builder object.builder add r1z1-127.0.0.1:6000/d1 100

- Then we add the next device (d2) to the builder files
- Continue this pattern for each device that is being added to the cluster.

Building the Rings

- Once all the devices have been added, let’s create the ring files. The rebalance command creates an actual ring file used by Swift to determine where data is placed.

- Once the rings are created, they need to be copied to the /etc/swift directory of every node in the cluster. No other action will be needed after that, because Swift automatically detects new ring data every 15 seconds.

- When a node that was down comes back online, be sure to provide any new ring files to it. If the node has a different (old) ring file, it will think that data isn’t where it should be and will do its part to move it back to where it’s “supposed” to be.

Building the Rings

- Here we run the commands to create the four rings:
  cd /etc/swift
  Swift-ring-builder account.builder rebalance
  Swift-ring-builder container.builder rebalance
  Swift-ring-builder object.builder rebalance
- As the command runs, each device signs up for its share of the partitions. During the build process, each device declares its relative weight in the system. The rebalancing process takes all the partitions and assigns them to devices, making sure that each device is subscribed according to its weight. That way a device with a weight of 200 gets twice as many partitions as a device with a weight of 100.
- Once the rebalance is complete, you should see the following additional files in /etc/swift:
  - account.ring.gz
  - container.ring.gz
  - object.ring.gz
- Copy these files to the /etc/swift directory of every node in the cluster.

Configuring Swift Logging

- Next, we will configure RSyslog to start logging for Swift. In this example, we'll direct all the log files to a single location.
- Swift uses RSyslog to help it manage where log messages go. In this example, we will set all the server processes to go to a single log for Swift.
- In each configuration file there is a setting called log_name . By default, all are set to swift .
- Additionally, you can configure external logging and alerting.

Creating the Log Configuration File

- Create a configuration file named 0-swift.conf in the /etc/rsyslog.d directory. It will contain one line:
  local0.* /var/log/swift/all.log
- Since we just created a script that will tell the system to log the all.log file in the directory /var/log/swift, we will need to create that directory and set the correct permissions on it.
- This command will create the directory the log files will be created in:
  mkdir /var/log/swift
- You also need to set permissions on the directory so the log process can write to it. For instance, the following commands do this on Ubuntu:
  chown -R syslog.adm /var/log/swift
  chmod -R g+w /var/log/swift

Restarting Rsyslog to Begin Swift Logging

- To start logging with the new Swift settings, restart RSyslog with the following command:
  - service rsyslog restart

Configuring a Proxy Server

- The proxy server process, is the connection between your Swift cluster and the outside world. This server process:
  - Accepts incoming HTTP requests
  - Looks up locations on the rings
  - Interacts with the authentication/authorization middleware
  - Forwards requests to the right account, container, or object server
  - Returns responses to the originating requester

Setting the Hash Path Prefix and Suffix

- The first configuration change to make is to set arbitrary, hard-to-guess strings as the swift_hash_path_prefix and swift_hash_path_suffix settings in /etc/swift/swift.conf.

- You add these strings to the pathnames in order to prevent a denial-of-service (DOS) attack. If someone knows the hash path suffix and prefix, he could determine the actual partition where objects would be stored. An attacker could generate containers and objects with that partition and repeatedly put large files to the same partition until the drive was full.

- Edit the swift.conf file in the /etc/swift/ directory and add a unique string to swift_hash_path_prefix and swift_hash_path_suffix :

Setting the Hash Path Prefix and Suffix

```
# swift_hash_path_suffix and swift_hash_path_prefix are used as part of the
# the hashing algorithm when determining data placement in the cluster.
# These values should remain secret and MUST NOT change
# once a cluster has been deployed.
[swift-hash]
swift_hash_path_suffix = RzUfDdu32L7J2ZBDYgsD6YI3Xie7hTVO8/oaQbpTbI8=
swift_hash_path_prefix = OZ1uQJNjJzTuFaM8X3v%fsJ1iR#F8wJjf9uhRiABevQ4
```

- We recommend selecting random strings of at least 32 characters for the prefix and suffix, which should be different.
- You can generate a 32-character string fairly easily using a number of different methods. One option might be to run:
  - head -c 32 /dev/random | base64
- Once you have two different strings saved, be sure to keep them secret. These make the partition path reasonably unguessable.

Starting the Proxy Server

- Now start up the proxy server process:

  swift-init proxy start

  Starting proxy-server...(/etc/swift/proxy-server.conf)
- If you get an error ending with KeyError: getpwnam(): name not found: swift it means that a user named swift hasn’t been created.

Setting up TempAuth Authentication and Authorization with Swift

- How to start memcached
- TempAuth lets you specify accounts in configuration files.
- memcached , short for “memory cache daemon,” is a process that stores data in memory for fast retrieval.
- The TempAuth middleware stores tokens in Memcache. If the memcached process is not running, tokens cannot be validated, and accessing Swift generally becomes impossible.
- On non-Debian-based Linux distributions, you need to ensure that memcached is running:
  - service memcached start
  - chkconfig memcached on
- For checking in ubuntu
  - ps aux | grep memcached

Adding Users to proxy-server.conf

- If you look in the file /etc/swift/proxy-server.conf, you can find the section that describes TempAuth:

```
[filter:tempauth]
use = egg:swift#tempauth
# You can override the default log routing for this filter here:
...
# <account> is from the user_<account>_<user> name.
# Here are example entries, required for running the tests:
user_admin_admin = admin .admin .reseller_admin
user_test_tester = testing .admin
user_test2_tester2 = testing2 .admin
user_test_tester3 = testing3
```

- With TempAuth, users are defined in the configuration file itself with the format:

```
user_$SWIFTACCOUNT_$SWIFTUSER = $KEY [group] [group] [...]
[storage_url]
```

Adding Users to proxy-server.conf

- You can create a user and account of your own, by adding a line under the default TempAuth account with the correct information:

  ```
  ...
  user_test_tester3 = testing3
  user_myaccount_me = secretpassword .admin .reseller_admin
  ```

- An optional parameter for the user entry would be to explicitly add the storage URL with port 8080 to the end of the line. This might be needed in some cases, depending on your particular configuration:

  ```
  user_myaccount_me = secretpassword .admin .reseller_admin <storage URL:8080>
  ```

- Two configurations must also be set for the account. Locate and set the allow_account_management and account_autocreate options to true in the file:

  ```
  allow_account_management = true
  account_autocreate = true
  ```

Starting the Servers and Restarting the Proxy

- To access user accounts, the account server process must be started. Now is also a good time to start the container and object server processes.
- When the proxy server is unable to reach one of the other main server processes, it will return a 503 Service Unavailable .

```
swift-init account start
Starting account-server...(/etc/swift/account-server.conf)
swift-init container start
Starting container-server...(/etc/swift/container-server.conf)
swift-init object start
Starting object-server...(/etc/swift/object-server.conf)
```

- Then, because we have changed the proxy server process configuration, we will restart the proxy server process:

```
swift-init proxy restart
Signal proxy-server pid: 5240 signal: 15
proxy-server (5240) appears to have stopped
Starting proxy-server...(/etc/swift/proxy-server.conf)
```

Account Authentication

- Now let's authenticate the account that was added to TempAuth in Swift using cURL:
  - curl -v -H 'X-Auth-User: $SWIFTACCOUNT:$SWIFTUSER' -H 'X-Auth-Key: <password>' <AuthURL>
- Sending in the authentication:
  - curl -v -H 'X-Auth-User: myaccount:me' -H 'X-Auth-Key: secretpassword' http://localhost:8080/auth/v1.0/
- You should get a response similar to:

```
About to connect() to localhost port 8080 (#0)
Trying ::1... Connection refused
Trying 127.0.0.1... connected
Connected to localhost (127.0.0.1) port 8080 (#0)
GET /auth/v1.0/ HTTP/1.1
User-Agent: curl/7.19.7 (i486-pc-linux-gnu) libcurl/7.19.7 OpenSSL/0.9.8kzlib/1.2.3.3 libidn/1.15
Host: localhost:8080
Accept: */*
X-Auth-User: myaccount:me
X-Auth-Key: secretpassword
HTTP/1.1 200 OK
X-Storage-Url: http://127.0.0.1:8080/v1/AUTH_myaccount
X-Storage-Token: AUTH_tk265318ae5e7e46f1890a441c08b5247f
X-Auth-Token: AUTH_tk265318ae5e7e46f1890a441c08b5247f
X-Trans-Id: txc75adf112791425e82826d6e3989be4d
Content-Length: 0
Date: Tue, 21 Mar 2013 22:48:40 GMT
```

Account Authentication

- If the connection is refused, verify that it was attempting to connecton port 8080 of the localhost address.
- The HTTP response contains the X-Auth-Token :

  < X-Auth-Token: AUTH_tk265318ae5e7e46f1890a441c08b5247f
- It will be stored in Memcache so that future storage requests can authenticate that token.
- Again, to take some mystery out of what we are creating, let's open an interactive Python session and show what the entry in Memcache looks like:

```
python
>>> import swift.common.memcached as memcached
>>> memcache = memcached.MemcacheRing(['127.0.0.1:11211'])
>>> print memcache.get('AUTH_/user/myaccount:me')
AUTH_tk58ad6d3ca1754ca78405828d72e37458
>>> print memcache.get(
...
'AUTH_/token/AUTH_tk58ad6d3ca1754ca78405828d72e37458')
[1394158570.037054, 'myaccount,myaccount:me,AUTH_myaccount']
```

Verifying Account Access

- Now you're ready to make your first request. Here we will attempt to list the containers in the account. The system should return a response of 204 No Content because we have not created any containers yet:

```
curl -v -H 'X-Storage-Token: AUTH_tk58ad6d3ca1754ca78405828d72e37458'
http://127.0.0.1:8080/v1/AUTH_myaccount/

* About to connect() to 127.0.0.1 port 8080 (#0)
*Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /v1/AUTH_myaccount HTTP/1.1
> User-Agent: curl/7.19.7 (i486-pc-linux-gnu) libcurl/7.19.7 OpenSSL/0.9.8k
zlib/1.2.3.3 libidn/1.15
> Host: 127.0.0.1:8080
> Accept: */*
> X-Storage-Token: AUTH_tk215c5706a61048c09819cd6ba60142ef
>
< HTTP/1.1 204 No Content
< X-Account-Object-Count: 0
< X-Account-Bytes-Used: 0
< X-Account-Container-Count: 0
< Accept-Ranges: bytes
< X-Trans-Id: txafe3c83ed76e46d2a9536dd61d9fcf09
< Content-Length: 0
< Date: Tue, 21 Jun 2011 23:23:23 GMT
```

Congratulations

you've created an account in Swift!

Creating a Container

- Accounts and containers are just SQLite databases. For each account, there is an account database that contains a listing of all the containers in that account.
- For each container, there is a container database that stores a listing of all of the objects in that container.
- Each of these databases is stored in the cluster and replicated in the same manner as the objects.
- Using the auth token from before, run the following command to create a container called mycontainer:
  curl -v -H 'X-Storage-Token: AUTH_tk58ad6d3ca1754ca78405828d72e37458' -X PUT http://127.0.0.1:8080/v1/AUTH_myaccount/mycontainer

- You should see something similar to:
  201 Created
  Success!

Creating a Container

- Here's what is happening behind the scenes:
  - The proxy server process sends a request to the account server process to updatethe account database with a new container
  - The proxy server process sends a request to the container server process to create the container database record
- What's interesting here is that these account/container databases are simply SQLite databases. In the following example we open the path to one of those account databases with SQLite, list the tables, select the account_stat table, and view the contents. Because we only have one container, it will return the single row with information about the container:

```
sqlite3 /srv/node/d2/accounts/497/e15/7c7d7a8558f1774e7f06d95094136e15/7c7d7a8558f1774e7f06d95094136e15.db
sqlite> .tables
account_stat    container    incoming_sync    outgoing_sync    policy_stat
sqlite> select * from account_stat;
AUTH_admin|1308715316.66344|1308715316.64684|0|0|0|0|00000000000000000000000000000000|ccfa951a-82a5-42fc-96c1-7c3e116e6e2e||0|
sqlite> .quit
```

Uploading an Object

- Now let's use the Swift command-line interface to upload a file. The command will look like this:

  swift -A <AUTH_URL> -U <account:username> <containername> <filename>
- Notice that the verb is upload , but remember that it is still a PUT request when the swift command sends it to the storage URL. Also notice that the container and file names are at the end of the request, separated by a space.
- Let's try to upload a file:

  swift -A http://127.0.0.1:8080/auth/v1.0/ -U myaccount:me -K secretpassword upload mycontainer some_file

Downloading an Object

```
swift -A http://127.0.0.1:8080/auth/v1.0/ -U
myaccount:me -K secretpassword download
mycontainer some_file
```

Starting the Consistency Processes

- Many consistency processes need to run to ensure that Swift is working correctly.
  - Configuring and turning on rsync
  - Starting the replicator processes
  - Starting the other consistency processes

Configuring rsync

- On Ubuntu:
- Edit or create the etc/default/rsync file; add RSYNC_ENABLE=true on one line:
  RSYNC_ENABLE=true
- Edit or create the /etc/rsyncd.conf file, adding in the following Swift information:

```
uid = swift
gid= swift
log file = /var/log/rsyncd.log
pid file = /var/run/rsyncd.pid

[account]
max connections = 25
path = /srv/node/
read only = false
lock file = /var/lock/account.lock

[container]
max connections = 25
path = /srv/node/
read only = false
lock file = /var/lock/container.lock

[object]
max connections = 25
path = /srv/node/
read only = false
lock file = /var/lock/object.lock
```

Configuring rsync

- Now it is time to start the rsync service on Ubuntu:
  - service rsync start
- After the restart, you should test that rsync is running with these debugging commands:
  - rsync localhost::
  - rsync localhost::account
  - rsync localhost::container
  - rsync localhost::object

Starting the Remaining Consistency Processes

- Consistency services can be started individually for a process. Start the replicators as follows:
  - swift-init account-replicator start
  - swift-init container-replicator start
  - swift-init object-replicator start
- Alternatively, the consistency services can be started all at once using the the swiftinit command with an all option:
  - swift-init all start
- Running the command as this point will start up the auditor, the updater, and the account reaper. If we had not already started the replicators, it would have started them as well.
- At this point, all the Swift processes should be running and OpenStack Swift should be fully installed.

Congratulations!

- **Installation is half the battle and if you have followed along this far, you likely have your very own Swift cluster up and running.**

- We hope going through this installation has given you some perspective and understanding of the parts that cooperate to make Swift work.

Have A Nice Day :)