# Lecture 8

Neutron, Data,
Restful Interface,
Restful Python Interface

# OpenStack Internals

The primary sub-parts of the OpenStack Cloud software:

| | Service | Comments |
|---|---|---|
| nova | Compute | Manages pools of computer resources |
| glance | Image | Disk and server images |
| keystone | Identity | Common authentication system |
| horizon | Dashboard | GUI |
| cinder | Block Storage | Persistent block storage devices |
| swift | Object Storage | Object and file management |
| neutron | Networking | IP addresses, VLANS, etc. |
| trove | Database | Provisions relational and non-relational DBs |
| heat | Orchestration | Launches composite cloud applications based on templates, specify relationships between resources |
| ironic | Bare metal provisioning | Provisions and turns on and off bare metal machines instead of virtual machines |
| ceilometer | telemetry | Billing system |
| sahara | Elastic Map Reduce | Handles Hadoop clusters |
| zaqar | Multiple Tenant Cloud Messaging | Can be used in SaaS to send messages between multiple components of the SaaS |

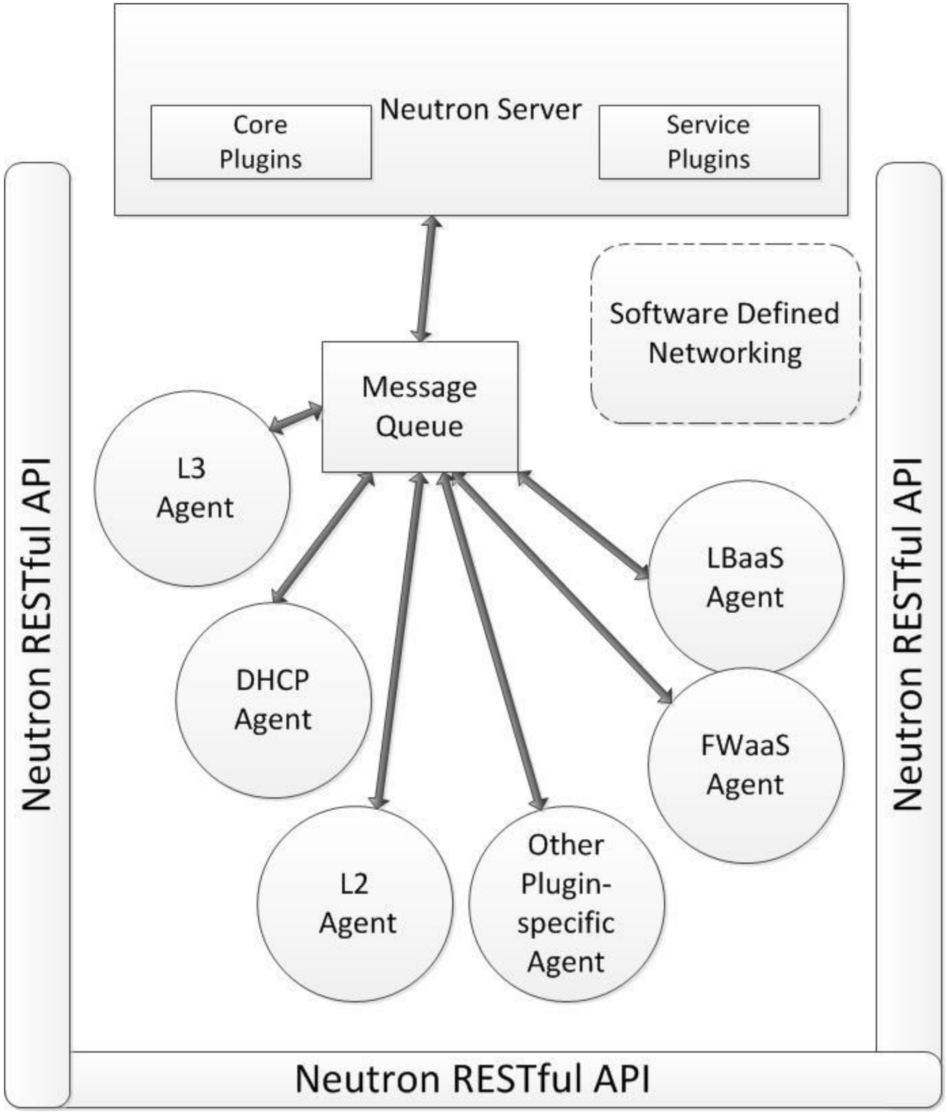
# OpenStack Neutron

- Neutron provides considerable functionality that was not provided by nova-network.
- Neutron allows users to configure their own network topology including:
  - multi-tier networks (web tier, application tier),
  - their own network (how to assign IP addresses, etc.)
- It also allows use of advanced network services, including services intended to improve security and quality of service.

# OpenStack Neutron (cont'd)

- When Neutron is being used (and not nova-network), there is a portion of Neutron that runs on the controller node as a python daemon
  - It provides network APIs and passes messages to Neutron plugins

# OpenStack Neutron (cont'd)--Overview



# OpenStack Neutron (cont’d)

- OpenStack Neutron components communicate with each other using AMQP and a message queue.
- Neutron uses various plugin agents to implement different protocols or functions. There are two kinds of plugins:
  - Core plugins
  - Service plugins

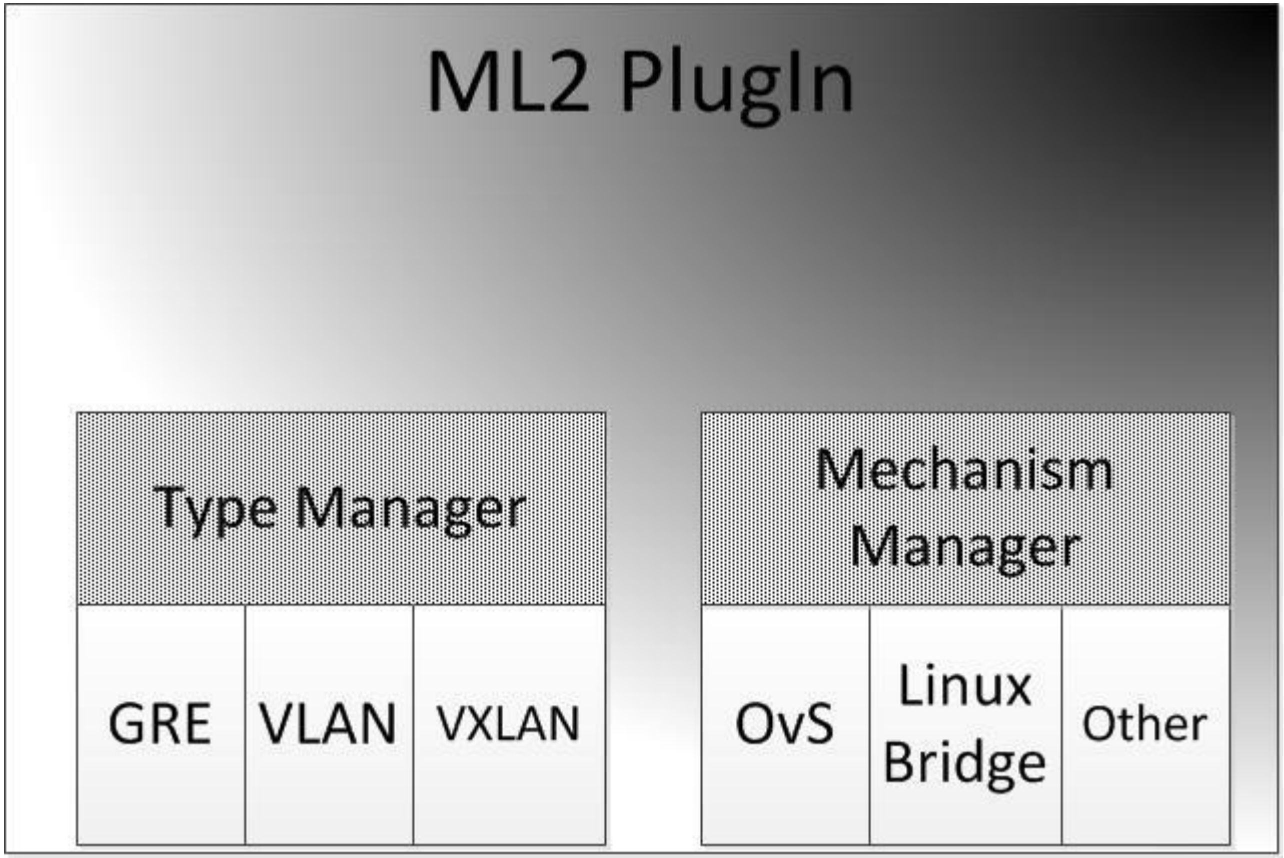
# OpenStack Neutron (cont'd)

- Core plugins implement basic network management including
  - IP address management (DHCP agent) and
  - L2 networking (ML2 plugin),
- Service plugins implement additional services including (among others):
  - load balancing (Load Balancing as a Service or LBaaS)
  - VPNs (VPNaaS)
  - firewalls (FWaaS)
  - metering

# OpenStack Neutron (cont’d)—ML2 Plugin

- The Modular Layer 2 (ML2) plugin is a core Neutron plugin that allows Neutron to work with different kinds of layer 2 networks.
- ML2 has *types* and *mechanisms*:
  - *type* is the type of network being used
  - the *mechanism* is the technology being used to implement the network

# OpenStack Neutron (cont’d)—ML2 Plugin (cont’d)



ML2 PlugIn

Type Manager

GRE

VLAN

VXLAN

Mechanism Manager

OvS

Linux Bridge

Other

# OpenStack Neutron (cont’d)—ML2 Plugin

- In the previous figure, we see three kinds of network types listed (other kinds can be added), these are:
  - GRE (Generic Routing Encapsulation)
  - VLAN (Virtual Local Area Network)
  - VXLAN (Virtual Extensible Local Area Network)

# OpenStack Neutron (cont’d)—ML2 Plugin

- Generic Routing Encapsulation (GRE) is a tunneling protocol used across an IP network to encapsulate many other protocols.
  - Tunneling protocols are often used to allow a different (foreign) protocol to run over a network that does not support that particular protocol
  - They are also often used to provide services that the underlying network does not support

# OpenStack Neutron (cont’d)—ML2 Plugin

- A Virtual Local Area Network allows partitioning of a network such that users (in this case, VMs) on one VLAN cannot see the traffic on a different VLAN

# OpenStack Neutron (cont’d)—ML2 Plugin

- In regard to ML2 mechanisms, OvS stands for Open vSwitch:
  - Open vSwitch is a well known open source method for creating virtual switches to connect to VMs in a software defined network
  - Open vSwitch employs Openflow, which is a software defined network standard
  - Linux bridges are an alternative method to create a virtual switch to connect to a VM

# OpenStack Neutron (cont’d)—ML2 Plugin

- The purpose of a Virtual Extensible Local Area Network is to improve scalability of networking in the cloud by extending the address space available to create VLANs.
- To do this, it overlays a layer 2 network on top of a layer 3 network.
- Like a VLAN, a VM on a VXLAN can only see other VMs on that VXLAN, they can’t see the traffic on other VXLANs.

# OpenStack Neutron (cont'd)—ML2 Plugin

- Note that Neutron allows connecting to full blown SDN controllers, such as Open Daylight or Floodlight
  - SDN controllers can be used to provide a view of the complete network topology and to monitor the network state, both physical and virtual.
  - SDN controllers can also manage changes to the SDN.
- This can be done either as a separate service plugin, or through the ML2 plugin (or both).

# OpenStack Internals

The primary sub-parts of the OpenStack Cloud software:

| | Service | Comments |
|---|---|---|
| nova | Compute | Manages pools of computer resources |
| glance | Image | Disk and server images |
| keystone | Identity | Common authentication system |
| horizon | Dashboard | GUI |
| cinder | Block Storage | Persistent block storage devices |
| swift | Object Storage | Object and file management |
| neutron | Networking | IP addresses, VLANS, etc. |
| trove | Database | Provisions relational and non-relational DBs |
| heat | Orchestration | Launches composite cloud applications based on templates, specify relationships between resources |
| ironic | Bare metal provisioning | Provisions and turns on and off bare metal machines instead of virtual machines |
| ceilometer | telemetry | Billing system |
| sahara | Elastic Map Reduce | Handles Hadoop clusters |
| zaqar | Multiple Tenant Cloud Messaging | Can be used in SaaS to send messages between multiple components of the SaaS |

# OpenStack Data Storage

- There are three kinds of storage with OpenStack:
  - Ephemeral
  - Block Storage
  - Object Storage

# OpenStack Data Storage (cont’d)

- Each running Virtual Machine receives some ephemeral storage,
  - it’s used to store the operating system of the image, and local data
  - The lifetime of the data in this storage is the lifetime of the Virtual Machine:
    - that is, the ephemeral data is no longer preserved when the Virtual Machine is terminated.

# OpenStack Data Storage (cont’d)

- Block storage is performed by Cinder,

- Object Storage is performed using Swift.

# OpenStack Data Storage (cont'd)--Cinder

- Cinder was originally part of Nova (called nova-volume).
- Cinder attaches storage volumes to a virtual machine
- A storage volume is a storage area with its own file system
  - The server that is running Cinder can have storage volumes located on physical disks attached to the Cinder server itself, or can be located on other physical disks.

# OpenStack Data Storage (cont’d)—Cinder (cont’d)

- Cinder has three basic services:
  - cinder-scheduler
    - schedules the appropriate volume service
  - cinder-volume
    - manages block storage devices—does load balancing between volumes
  - cinder-backup
    - allows backup of volumes

# OpenStack Data Storage (cont’d)—Swift

- OpenStack Swift stores data as binary objects (can have associated metadata) that are retrieved and written using HTTP commands (GET, PUT, etc.)
  - originally created by Rackspace, and part of the original OpenStack Austin release
- Swift stores objects on object servers.
- Swift is best used for storing unstructured data, such as email, images, audio, and video, etc.

# OpenStack Data Storage (cont’d)—Swift (cont’d)

- Swift organizes object data into a hierarchy.
- Your account is the top level of the hierarchy, and you own the information associated with that account (project or tenant).
- Inside the account (project or tenant) can be containers (namespace for objects).
- Within one account, the names of containers must be unique.
- The containers may have associated Access Control Lists (ACLs) that specify who can have access to the objects within the container (individual objects do not have associated ACLs).

# OpenStack Data Storage (cont'd)—Swift (cont'd)

- Swift has the concept of a “ring”.
- A ring keeps track of where data is stored in the cluster
  - it is used to map names to the physical locations of objects (physical device on physical node)
- There are rings for:
  - accounts (projects or tenants)
  - containers
  - each separate storage policy

# OpenStack Data Storage (cont’d)—Swift (cont’d)

- For data backup, rings are divided into zones, within which data is replicated.
- By default, three replicas of data objects are created and stored in a separate zone (to protect against hardware failures).
- A zone can consist of a single disk drive or a server in either the local data center or a different data center

# OpenStack RESTful Interface

- There are several different ways to access the OpenStack RESTful interface.  Four of the most popular are:

1) the OpenStack Dashboard..
2) using cURL to send appropriate HTTP commands
3) using python with the python libraries provided with OpenStack
4) use the command line clients.

# OpenStack RESTful Interface

- Different OpenStack RESTful interfaces listen at different port numbers on the OpenStack url
  - See Table 13.10 text for some important OpenStack port numbers
- There are two versions of the Keystone interface commonly used today, version 2 and version 3.
  - Version 3 keystone url:

    http://localhost:5000/v3

  - Version 2 keystone url:

    http://localhost:5000/v2.0

# OpenStack RESTful Interface

- Different OpenStack RESTful interfaces listen at different port numbers on the OpenStack url.
- There are two versions of the Keystone interface commonly used today, version 2 and version 3.
  - Version 3 keystone url:

    http://localhost:5000/v3

  - Version 2 keystone url:

    http://localhost:5000/v2.0

# OpenStack RESTful Interface

- In the following slides, we will see one example of how to authenticate using each one of the different ways to access the OpenStack RESTful interface

- For how to start instances, etc., see the textbook and its associated code

# OpenStack RESTful Interface—cURL

**Keystone authentication using cURL command:**

```
curl -i -H "Content-Type: application/json" \
  -d @my_credentials \
  http://localhost:5000/v3/auth/tokens
```

# OpenStack RESTful Interface Interface—cURL

**Credentials file in JSON format**

```
{
"auth": {
    "identity": {
        "methods": [
            "password"
         ],
         "password": {
             "user": {
            "name": "admin",
             "domain": {
                "id": "default"
            },
            "password":"mypassword"
              }
         }
       }
 }
}
```

# OpenStack RESTful Interface Interface—python

**Keystone Authentication in python**

```
from keystoneauth1.identity import v3

from keystoneauth1 import session

from keystoneclient.v3 import client
auth = v3.Password(
        user_domain_name='default',
        username='admin',
        password='mypassword',
        project_domain_name='default',
        project_name='admin',
        auth_url='http://localhost:5000/v3'
        )
mysess=session.Session(auth=auth)

keystone=client.Client(session=mysess)

return keystone
```

# OpenStack RESTful Interface Interface—Common Client

- Using the Keystone version 3 API
  - To authenticate, set the environment variables as follows:

```
export OS_AUTH_URL=http://localhost:5000/v3
export OS_USERNAME=admin
export OS_PASSWORD=mypassword
export OS_USER_DOMAIN_NAME=default
export OS_IDENTITY_API_VERSION="3"
export OS_PROJECT_NAME="admin“
```

Then type:

```
openstack image list
```

# OpenStack RESTful Interface Interface—Separate Keystone Client

**Assuming you're using a devstack install:**

In the devstack directory is a file called openrc.
Type:

```
source openrc
```

This will set various environment variables to default values so that you can access the openstack stuff. Alternately, set various variables, as follows:

```
  export OS_USERNAME=admin
  export OS_PASSWORD=mypassword
  export OS_TENANT_NAME=admin
  export OS_AUTH_URL=http://localhost:5000/v2.0
```

(or whatever IP address has been internally assigned.)

(this is the keystone port on localhost, port 5000 tells us that. It's using version 2.0 of the keystone API)

```
  export OS_REGION_NAME=RegionOne
```

# OpenStack RESTful Interface Interface—Separate Keystone Client

- Since you set yourself up as the admin user, you can do some of the following commands:

  - To view all tenants (projects):
  - keystone tenant-list

  - To display all current users:
  - keystone user-list

# OpenStack RESTful Interface Interface—Separate Keystone Client

- Since you set yourself up as the admin user, you can do some of the following commands:

  - To view all tenants (projects):
  - keystone tenant-list

  - To display all current users:
  - keystone user-list

# OpenStack Python Interface

To enter the python environment, type:

```
python
```

To exist the python environment, type;

```
Ctrl-D
```

# OpenStack Python Interface (cont’d)

Authentication plugins are a way to extend how OpenStack performs authentication

- OpenStack clients access these plugins in order to authenticate

Various standard “identity plugins” are provided with Keystone, this includes:

V2 identity plugins

V3 identity plugins

# OpenStack Python Interface (cont’d)

V2 identity plugins

- defined in module keystoneclient.auth.identity.v2
- Includes
  - PasswordMethod—uses username/password to authenticate
  - TokenMethod—uses existing token to authenticate
- auth_url must be keystone V2 root  (http://url:5000/v2.0)

A V2 request can employ only one authentication method.

# OpenStack Python Interface (cont’d)

V3 identity plugins:

- defined in module keystoneclient.auth.identity.v3
- A V3 request can contain multiple authentication methods:
  - Includes “AuthMethod” objects that are sent to V3 keystone interface:
    - PasswordMethod—uses username/password to authenticate
    - TokenMethod—uses existing token to authenticate
- auth_url must be keystone V3 root  (http://url:5000/v3)

# OpenStack Python Interface (cont’d)

The following method is used to pass an AuthMethod object to the Keystone auth plugin:

```
myauth= v3.Auth(
            auth_url=…v3 keystone root url,
            auth_method = ..the AuthMethod object…
            user_domain_name =…,
            project_domain_name=…,
            project_id=….
          )
```

# OpenStack Python Interface (cont’d)

Note that you can also access this plugin similarly to how a V2 plugin is accessed, by doing specific calls that employ only a single authentication method. For example:

- keystoneclient.auth.identity.v3.Password

  auth_url, username, password,

  other arguments )
- keystoneclient.auth.identity.v3.Token(

  auth_url, token,

  other arguments )

- “other arguments” includes: user_domain_id, project_domain_id, project_id, project_name, etc.

# OpenStack Python Interface (cont’d)

For example:

```
v3.Password(
          auth_url=…v3 keystone root url,
          username=…,
          password=…
           user_domain_name =…,
          project_domain_name=…,
          project_id=…
          etc.
 )
```

# OpenStack Python Interface (cont’d)

```
def authfunc():
        from keystoneauth1.identity import v3
        from keystoneauth1 import session
        from keystoneclient.v3 import client
        auth = v3.Password(user_domain_name='default', username='admin',password='secret',project_domain_name='default', project_name='admin', auth_url='http://localhost:5000/v3')
        sess=session.Session(auth=auth)
        keystone=client.Client(session=sess)
        return keystone
```

# OpenStack Python Interface (cont’d)

In the code from the previous page, this line imports the keystone python client:

```
from keystoneauth1.identity import v3
```

The following line imports the keystone session library:

```
from keystoneauth1 import session
```

and this line (further down in the code) creates a session using the given authorization (the authorization is stored in the “auth” object):

```
sess=session.Session(auth=auth)
```

This line imports the keystone client library for the v3 interface:

```
from keystoneclient.v3 import client
```

and this line creates a client that employs the previously created session:

```
keystone=client.Client(session=sess)
```

# OpenStack Python Interface (cont'd)

keystoneclient.v3.client does changes and queries by using managers. Some example managers:

- keystoneclient.v3.users.UserManager
  - Create, delete, get, list user(s), add/remove users to groups, check if user is in a group
- keystoneclient.v3.services.ServiceManager
  - Create, retrieve, list, update delete services on the server
- keystoneclient.v3.groups.GroupManager
  - Create, delete, get, list, update groups
- keystoneclient.v3.endpoints.EndpointManager
  - Create, retrieve, delete, list endpoints
- keystoneclient.v3.ec2.EC2Manager
  - Create access key/secret key pair
  - Delete access key/secret key pair
  - Retrieve access key/secret key pair for given access key
  - List access key/secret key pairs for a given user

# OpenStack Python Interface (cont’d)

openstack.session.Session manages:

- authenticator
- transport
- user profile

A session does the following:

- Inserts authentication tokens into requests
- Retrieves endpoints from the authenticator
- Determines service preferences
- Sends requests using the appropriate transport

# OpenStack Python Interface (cont'd)

Openstack.session.Session(profile, user-agent=…, various arguments)

So a session object is associated with a:

- profile
- User agent
  - A user agent string is used to identify a software agent (software acting on behalf of a user)
  - Could include operating system, application type, revision.

# OpenStack Python Interface (cont’d)

- A profile object contains user preferences for various services:
  - service name
    - Service type—compute, identity, object-store, etc.
    - Desired name of the service
  - Region associated with the service
  - Version of the service—could be V2, V3, etc.
  - Interface—interface associated with the specified service

# OpenStack Python Interface (cont’d)

The OpenStack python SDK works as follows:

- Connection methods (associated with the Connection interface) accept and return Resource objects (associated with the Resource interface)

- Your session, authentication, transport and profile are maintained by an instance of a Connection
  - Based on a particular user’s profile, various OpenStack services are available to be accessed by that users through the Connection

# OpenStack Python Interface (cont’d)

Resources can be used directly (as well as being accessed through the Connection interface)

A Resource object is associated with the REST API of each service

The Resource class supports:

- Create Read Update Delete operations on the REST APIs
- Calling HTTP GET, POST, PUT on the associated Session
- Creating URLs

# OpenStack Python Interface (cont'd)

Values sent to/received from the service are implemented as attributes (type openstack.resource.prop) on the Resource class:

- These attributes can include a type that can be validated when requests are received

The attribute base_path should correspond to the url of the resource

Other Resource attributes are checked before a request is made to the associated REST API:

- allow_create, allow_delete, allow_retrieve, allow_head, allow_list
- these each have values of true/false

# OpenStack Python Interface (cont'd)

A Proxy class is associated with each service

A Proxy class provides a high level interface for a user that:

- Manages a Session
- Works with lower level Resource objects

A Connection object provides a high level interface for a user that:

- Provides a Proxy object for each Resource object (associated with each service)
- Is “built on top” of a Session object