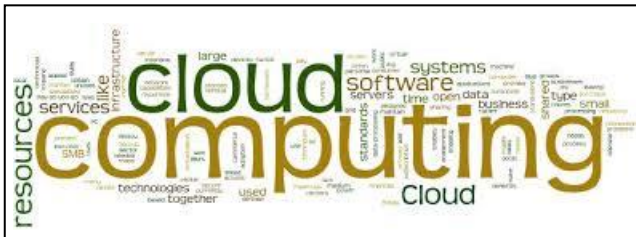


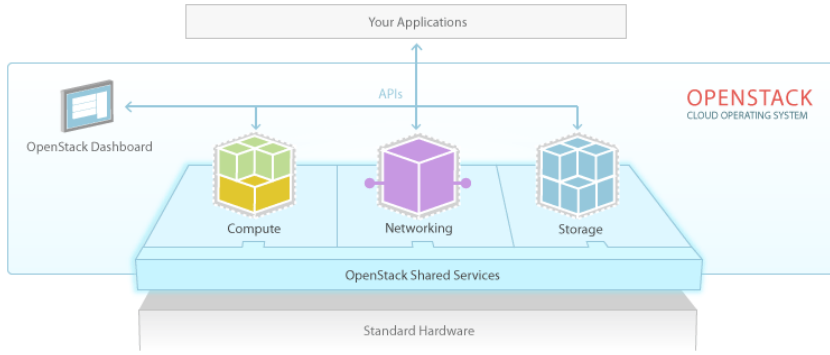
Part 2 – OpenStack: Architecture and Components

Gianluca Reali

Università degli Studi di Perugia

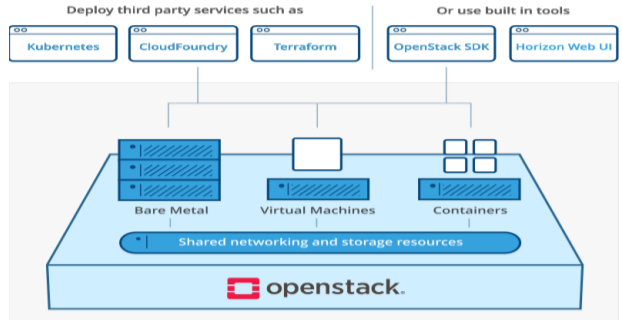
Ref. - Chap. 13 of the book “Introduction to Middleware” by Letha Hughes Etzkorn
- www.opestack.org

OpenStack



OpenStack is a cloud operating system that controls large **pools of compute, storage, and networking resources** throughout a datacenter. A **dashboard** that gives administrators control while empowering their users to provision resources through a **web interface**.

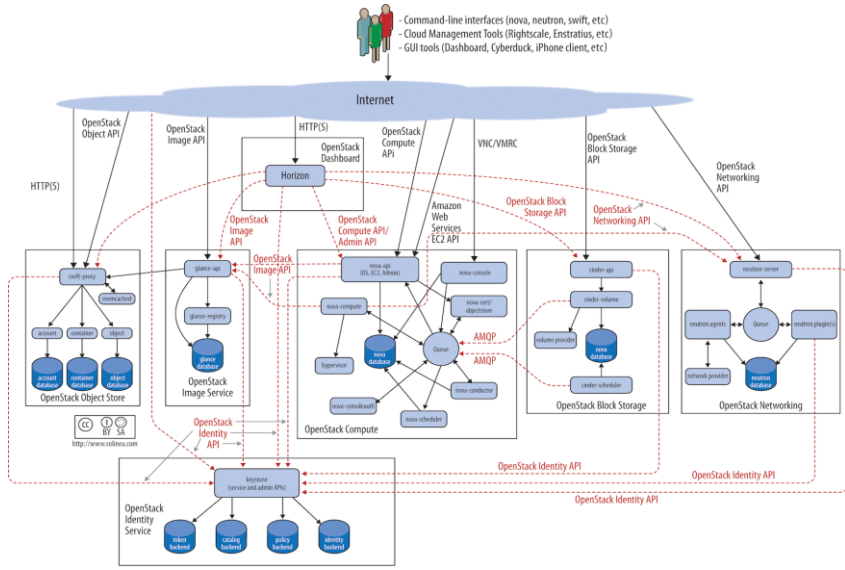
OpenStack



The OpenStack system consists of several key projects that you install separately. These projects work together depending on your cloud needs. These projects include **Compute**, **Identity Service**, **Networking**, **Image Service**, **Block Storage**, **Object Storage**, **Telemetry**, **Orchestration**, and **Database**. You can install any of these projects separately and configure them stand-alone or as connected entities.

OS Logical architecture

The following diagram shows the most common, but not the only possible, architecture for an OpenStack cloud:



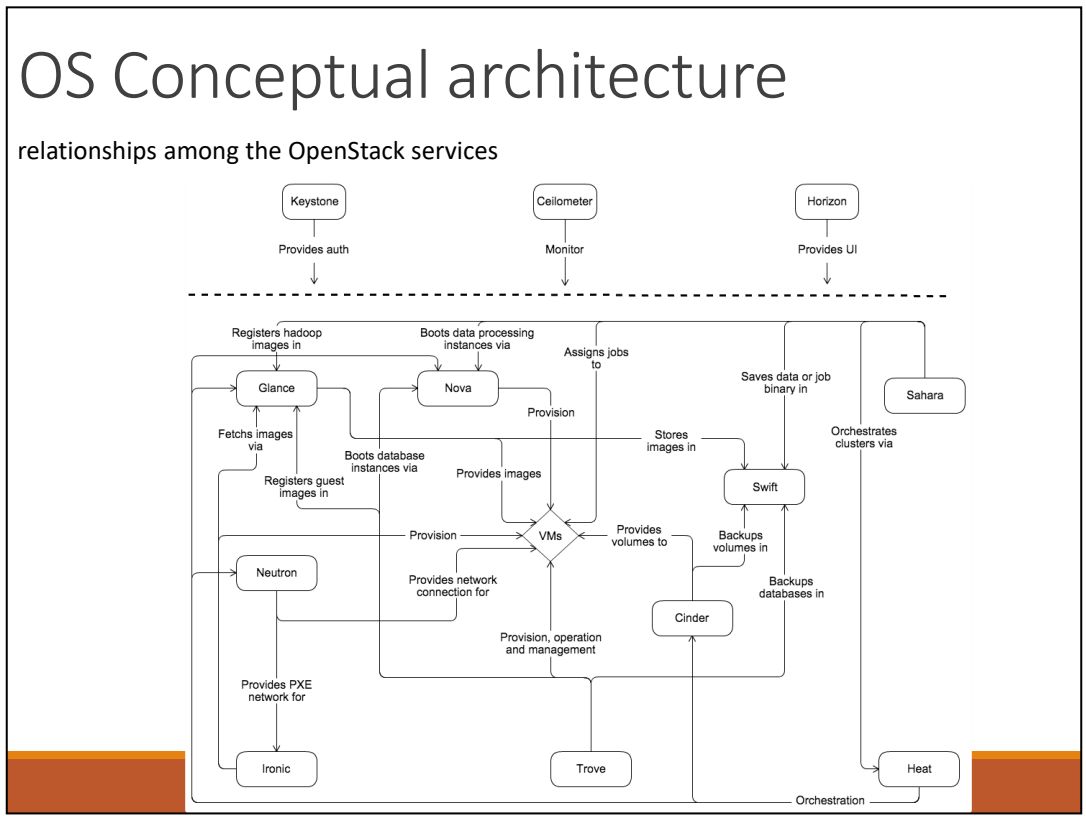
https://docs.openstack.org/install-guide/get-started-logical-architecture.html

To design, deploy, and configure OpenStack, administrators must understand the logical architecture. OpenStack modules are one of the following types:

Daemon Runs as a background process. On Linux platforms, a daemon is usually installed as a service. Script Installs a virtual environment and runs tests. Command-line interface (CLI) Enables users to submit API calls to OpenStack services through commands. OpenStack Logical Architecture shows one example of the most common integrated services within OpenStack and how they interact with each other. End users can interact through the dashboard, CLIs, and APIs. All services authenticate through a common Identity service, and individual services interact with each other through public APIs, except where privileged administrator commands are necessary.

Advanced Message Queuing Protocol (AMQP) è uno standard aperto che definisce un protocollo a livello applicativo per il message-oriented middleware. AMQP è definito in modo tale da garantire funzionalità di messaggistica, accodamento, routing (con paradigmi punto-punto e pubblicazione-sottoscrizione), affidabilità e sicurezza.

tanto li vedremo



https://docs.openstack.org/install-guide/get-started-conceptual-architecture.html

Trove is Database as a Service for OpenStack. It's designed to run entirely on OpenStack, with the goal of allowing users to quickly and easily utilize the features of a relational or non-relational

database without the burden of handling complex administrative tasks.

OpenStack bare metal provisioning a.k.a **Ironic** is an integrated OpenStack program which aims to provision bare metal machines instead of virtual machines, forked from the Nova baremetal driver.

The sahara project aims to provide users with a simple means to provision data processing frameworks (such as Apache Hadoop, Apache Spark and Apache Storm) on OpenStack. This is accomplished by specifying configuration parameters such as the framework version, cluster topology, node hardware details and more.

cose già dette

OS Conceptual architecture

- OpenStack consists of several independent parts, named the **OpenStack services**. All services authenticate through a **common Identity service**. Individual services interact with each other through **public APIs**, except where privileged administrator commands are necessary.
- Internally, OpenStack services are composed of **several processes**. All services have at least one API process, which listens for API requests, preprocesses them and passes them on to other parts of the service. With the exception of the Identity service, the actual work is done by distinct processes.

Message Oriented Middleware

- Openstack, as any other modern large-scale applications, is built as **distributed** network applications, with parts of the application in distinct processes and, possibly, in distinct parts of the world.
- All the same, the parts need to **work together** to behave as one reliable application. They need a way to communicate, and they must be able to tolerate failures.
- In principle, information could be shared through database or REST (e.g. HTTP), but it has some serious drawbacks.
  - A database is reliable, but it isn't designed to intermediate communication. Its focus is storing data, not moving it between processes.
  - REST helps you communicate efficiently, but it offers no reliability. If the party you're talking to is unavailable, the transmission is dropped.

Message Oriented Middleware

- The **store-and-forward messaging system** gives you efficient, reliable communication. Message brokers take responsibility for ensuring messages reach their destination, even if the destination is temporarily out of reach. Messaging APIs manage acknowledgments so that no messages are dropped in transit.
  - Remember Publisher-Subscriber pattern and queue-based Cloud Design Patterns

Message Oriented Middleware

- **Loosely Coupled Architecture:** OpenStack uses a *message queue* to coordinate operations and status information among services. The message queue service typically runs on the controller node.
- OpenStack projects use **AMQP**, an open standard for messaging middleware.
- The service's state is stored in a database. When deploying and configuring your OpenStack cloud, you can choose among several message broker and database solutions, such as RabbitMQ, MySQL, MariaDB, and SQLite.
- The default message queue service used is RabbitMQ.
- **RabbitMQ** broker stays between internal components of each service in OpenStack (e.g. nova-conductor, nova-scheduler in Nova service) and allow them to communicate each other in the loosely coupled way.

The Advanced Message Queuing Protocol (AMQP) is an open standard application layer protocol for message-oriented middleware. The defining features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security.

AMQP is a binary, application layer protocol, designed to efficiently support a wide variety of messaging applications and communication patterns. It provides flow controlled, message-oriented communication with message-delivery guarantees such as at-most-once (where each message is delivered once or never), at-least-once (where each message is certain to be delivered, but may do so multiple times) and exactly-once (where the message will always certainly arrive and do so only once), and authentication and/or encryption based on SASL and/or TLS. It assumes an underlying reliable transport layer protocol such as Transmission Control Protocol (TCP).

The AMQP specification is defined in several layers: (i) a type system, (ii) a symmetric, asynchronous protocol for the transfer of messages from one process to another, (iii) a standard, extensible message format and (iv) a set of standardised but extensible 'messaging capabilities.'

OpenStack Services/Projects

- **Dashboard /Horizon**
  - Provides a web-based self-service portal to interact with underlying OpenStack services, such as launching an instance, assigning IP addresses and configuring access controls.
- **Compute /Nova**
  - Manages the lifecycle of compute instances in an OpenStack environment. Responsibilities include spawning, scheduling and decommissioning of virtual machines on demand.
- **Networking /Neutron**
  - Enables Network-Connectivity-as-a-Service for other OpenStack services, such as OpenStack Compute. Provides an API for users to define networks and the attachments into them. Has a pluggable architecture that supports many popular networking vendors and technologies.

OpenStack Services/Projects

Shared services

- **Identity service /Keystone**
  - Provides an authentication and authorization service for other OpenStack services. Provides a catalog of endpoints for all OpenStack services.
- **Image service / Glance**
  - Stores and retrieves virtual machine disk images. OpenStack Compute makes use of this during instance provisioning.
- **Telemetry / Ceilometer**
  - Monitors and meters the OpenStack cloud for billing, benchmarking, scalability, and statistical purposes.

OpenStack Services/Projects

Storage services

- **Object Storage / Swift**
  - Stores and retrieves arbitrary unstructured data objects via a RESTful, HTTP based API. It is highly fault tolerant with its data replication and scale-out architecture. Its implementation is not like a file server with mountable directories. In this case, it writes objects and files to multiple drives, ensuring the data is replicated across a server cluster.
- **Block Storage / Cinder**
  - Provides persistent block storage to running instances. Its pluggable driver architecture facilitates the creation and management of block storage devices.

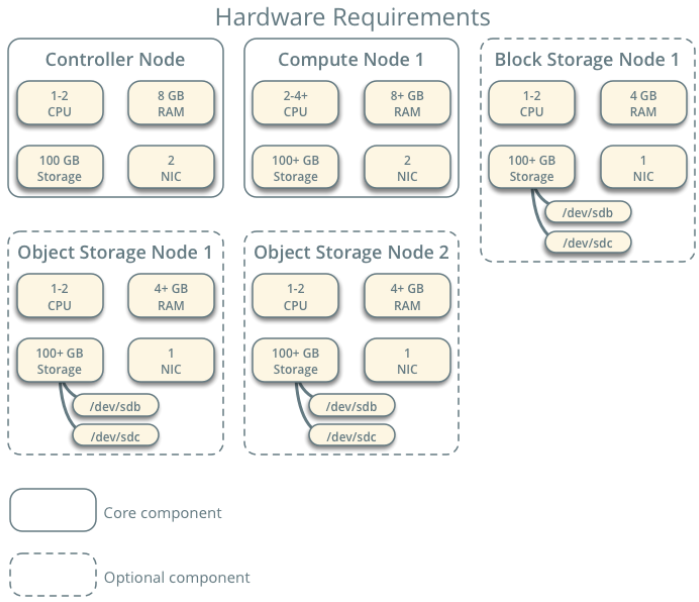
OpenStack Services/Projects

Higher-level services

Orchestration / Heat

- Orchestrates multiple composite cloud applications by using either the native HOT template format or the AWS CloudFormation template format, through both an OpenStack-native REST API and a CloudFormation-compatible Query API.

Example architecture



Example architecture

- **Controller:** The controller node runs the Identity service, Image service, management portions of Compute, management portion of Networking, various Networking agents, and the Dashboard. It also includes supporting services such as an SQL database, message queue, and NTP.

  Optionally, the controller node runs portions of the Block Storage, Object Storage, Orchestration, and Telemetry services.

  The controller node requires a minimum of two network interfaces.

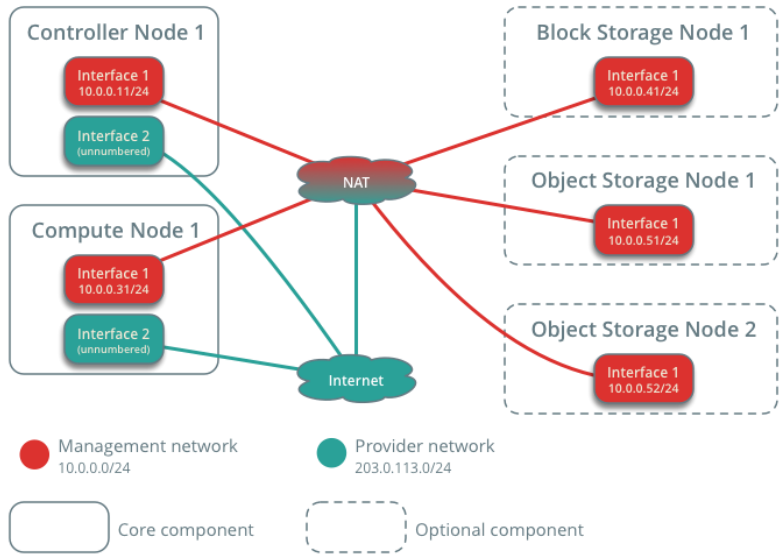
Example architecture

- **Compute:** The compute node runs the hypervisor portion of Compute that operates instances. By default, Compute uses the KVM hypervisor. The compute node also runs a Networking service agent that connects instances to virtual networks and provides firewalling services to instances via security groups.
  - You can deploy more than one compute node. Each node requires a minimum of two network interfaces.
- **Block Storage:** The optional Block Storage node contains the disks that the Block Storage and Shared File System services provision for instances.
  - Production environments should implement a separate storage network to increase performance and security.
  - You can deploy more than one block storage node. Each node requires a minimum of one network interface.

Example architecture

- **Object Storage:** The optional Object Storage node contain the disks that the Object Storage service uses for storing accounts, containers, and objects.
  - Production environments should implement a separate storage network to increase performance and security.
  - Each node requires a minimum of one network interface. You can deploy more object storage nodes.

Network Layout



Networking Alternatives

Networking¶

You can choose one of the following virtual networking options.

- **Networking Option 1: Provider networks:** The provider networks option deploys the OpenStack Networking service in the simplest way possible with primarily layer-2 (bridging/switching) services and VLAN segmentation of networks. Essentially, it bridges virtual networks to physical networks and relies on physical network infrastructure for layer-3 (routing) services. Additionally, a DHCP service provides IP address information to instances.
- **Networking Option 2: Self-service networks:** The self-service networks option augments the provider networks option with layer-3 (routing) services that enable self-service networks using overlay segmentation methods such as VXLAN. Essentially, it routes virtual networks to physical networks using NAT. Additionally, this option provides the foundation for advanced services such as LBaaS and FWaaS.

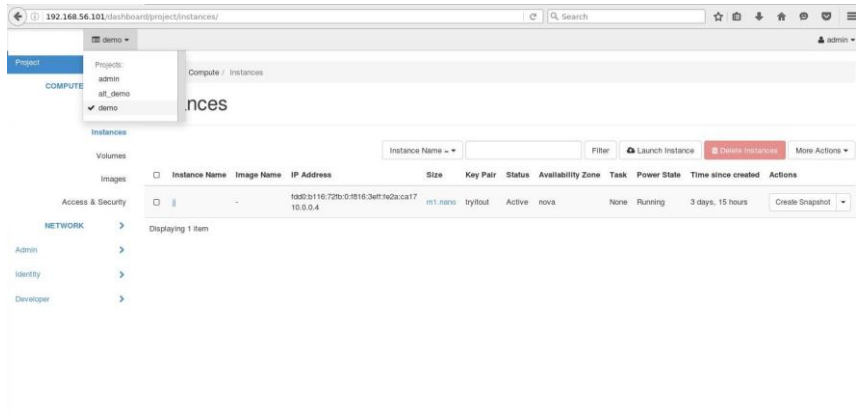
19

Load Balancer as a Service (LBaaS)

Firewall as a Service (FWaaS)

How to use OpenStack

- Basic install gives you a project and login named demo, and a project and login named admin.
- You can, of course, add additional projects.



How to use OpenStack (cont'd)

- An "**instance**" is what OpenStack calls a running virtual machine
- An **image** is a copy of an operating system.
- When an instance is started, it must be given some kind of OS to run
- OpenStack can run many different versions of Linux (Ubuntu, RedHat, Fedora, SUSE, Debian) and it can run Windows
- Several tested images are available or you can create your own image

How to use OpenStack (cont'd)

- For testing an instance of a cirros operating system can be used.
- The CirrOS OS is a test OS, that does very little but also doesn't require much memory to run:
  - cirros-0.3.2-x86_64-uec
- Using m1.nano size
- Using a previously created **keypair**

How to use OpenStack (cont'd)

- The **keypair** associated with an OpenStack instance consists of a public key and a private key
- You can use the OpenStack dashboard to create these, or you can create them other ways (such as with *ssh-keygen*)
- The public key must be registered with OpenStack
- You must download the private key to keep it safe:
  - You will use your private key later on to access your OpenStack instance

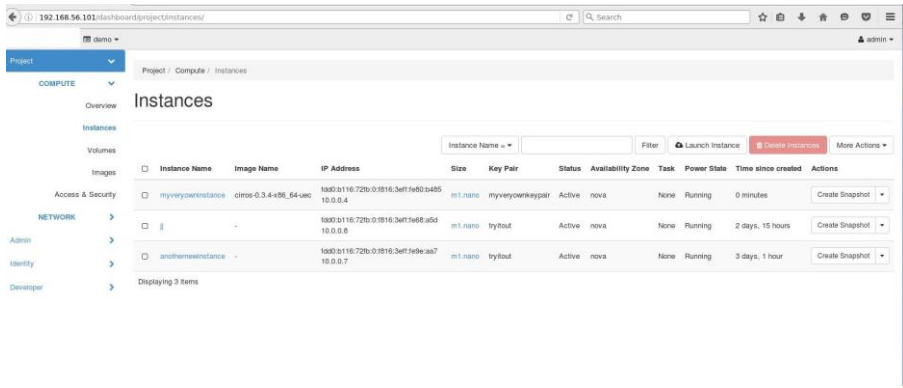
meh

How to use OpenStack (cont'd)

The **flavor** tab provides flavors that correspond to choices about:

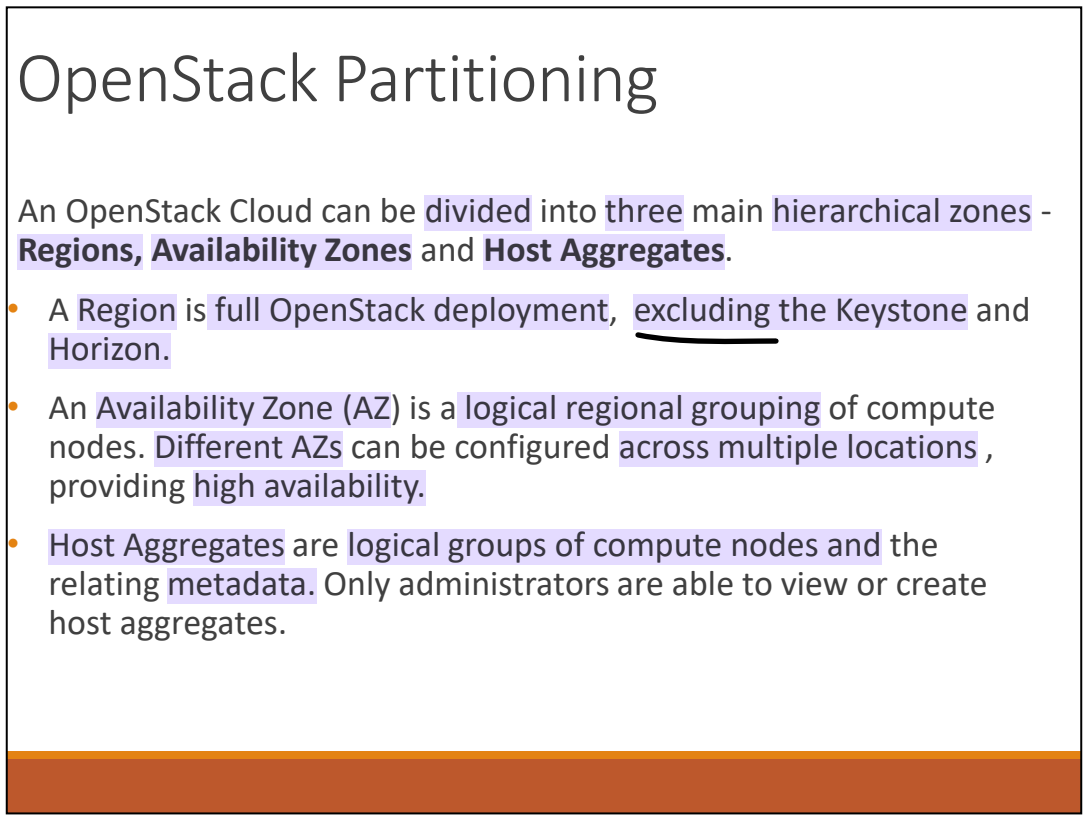
- Number of virtual CPUs
- How much disk memory (different kinds of disks)
- How much RAM

How to use OpenStack (cont'd)



How to use OpenStack (cont'd)

- Overcommitting CPU and RAM: OpenStack allows you to overcommit CPU and RAM on compute nodes. This allows you to increase the number of instances running on your cloud at the cost of reducing the performance of the instances. The Compute service uses the following ratios by default:
  - CPU allocation ratio: 16:1
  - RAM allocation ratio: 1.5:1



https://docs.openstack.org/keystone/latest/admin/identity-concepts.html

Region: An Identity service API v3 entity. Represents a general division in an OpenStack deployment. You can associate zero or more sub-regions with a region to make a tree-like structured hierarchy. Although a region does not have a geographical connotation, a deployment can use a geographical name for a region, such as us-east.

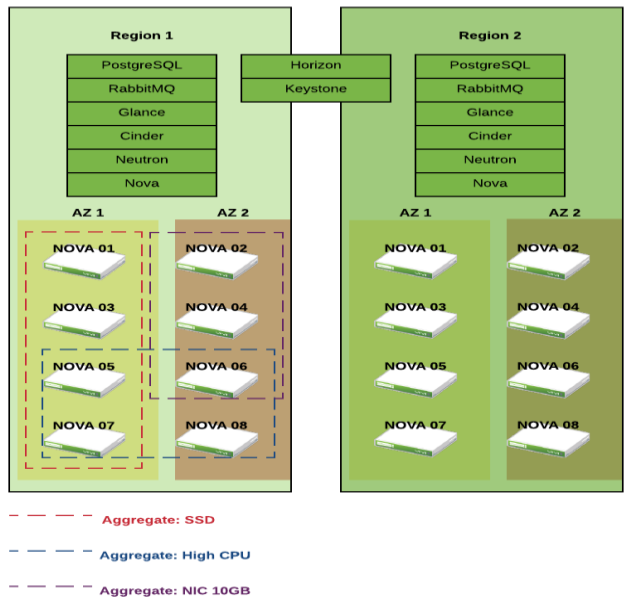
A Region is full OpenStack deployment, including its own API endpoints, networks and compute resources[1], excluding the Keystone and Horizon. Each Region shares a single set of Keystone and Horizon services. Each Region shares a single set of Keystone and Horizon services. The default OpenStack region name is RegionOne.

An aggregates metadata is commonly used to provide information for use with the Compute scheduler (for example, limiting specific flavors or images to a subset of hosts). Metadata specified in a host aggregate will limit the use of that host to any instance that has the same metadata specified in its flavor.

An Availability Zone (AZ) is a logical regional grouping of compute nodes. Different AZs can be configured across multiple locations , providing high availability. Unlike Host Aggregates, Availability Zones are exposed to end users

who can select a particular availability zone for VM placement while the instance is launched.

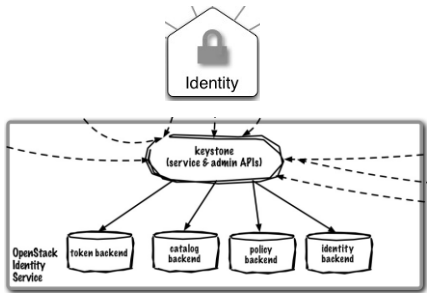
OpenStack Partitioning



https://docs.openstack.org/keystone/latest/admin/identity-concepts.html

OpenStack Identity Service: Keystone

Keystone is an OpenStack project that provides **Identity**, **Token**, **Catalog** and **Policy** services for use specifically by projects in the OpenStack family. It implements OpenStack's **Identity API**



- **Core use cases:**

  Installation-wide authentication and authorization to OpenStack services
- **Core key capabilities:**
  - Authenticate user / password requests against multiple backends (SQL, LDAP, etc) (Identity Service)
  - Validate / manage tokens used after initial username/password verification (Token Service)
  - Endpoint registry of available services (Service Catalog)
  - Authorize API requests (Policy Service)
    - Policy service provides a rule-based (RBAC) authorization engine and the associated rule management interface.

Image Source: http://www.solinea.com/2013/04/17/openstack-summit-intro-to-openstack-architecture-grizzly-edition/

Role-Based Access Control (RBAC)

Keystone Architecture

Services: Keystone is organized as a group of internal services exposed on one or many endpoints. Many of these services are used in a combined fashion by the frontend.

- **Identity Service**
- **Resource Service**
- **Assignment Service**
- **Token Service**
- **Catalog Service**
- **Policy Service**

The Resource Service (1/2)

The Resource service provides data about *projects* and *domains*.

- **Projects (Tenants)**: They represent the base unit of ownership in OpenStack, in that all resources in OpenStack should be owned by a specific project. A project itself must be owned by a specific domain, and hence all project names are **not** globally unique, but unique to their domain. If the domain for a project is not specified, then it is added to the default domain.
- **Domains**: They are a high-level container for projects, users and groups. Each is owned by exactly one domain. Each domain defines a namespace where an API-visible name attribute exists. Keystone provides a default domain, named '**Default**'.

31

https://docs.openstack.org/keystone/latest/admin/identity-concepts.html

Domain: An Identity service API v3 entity. Domains are a collection of projects and users that define administrative boundaries for managing Identity entities. Domains can represent an individual, company, or operator-owned space. They expose administrative activities directly to system users. Users can be granted the administrator role for a domain. A domain administrator can create projects, users, and groups in a domain and assign roles to users and groups in a domain.

Project: A container that groups or isolates resources or identity objects. Depending on the service operator, a project might map to a customer, account, organization, or tenant.

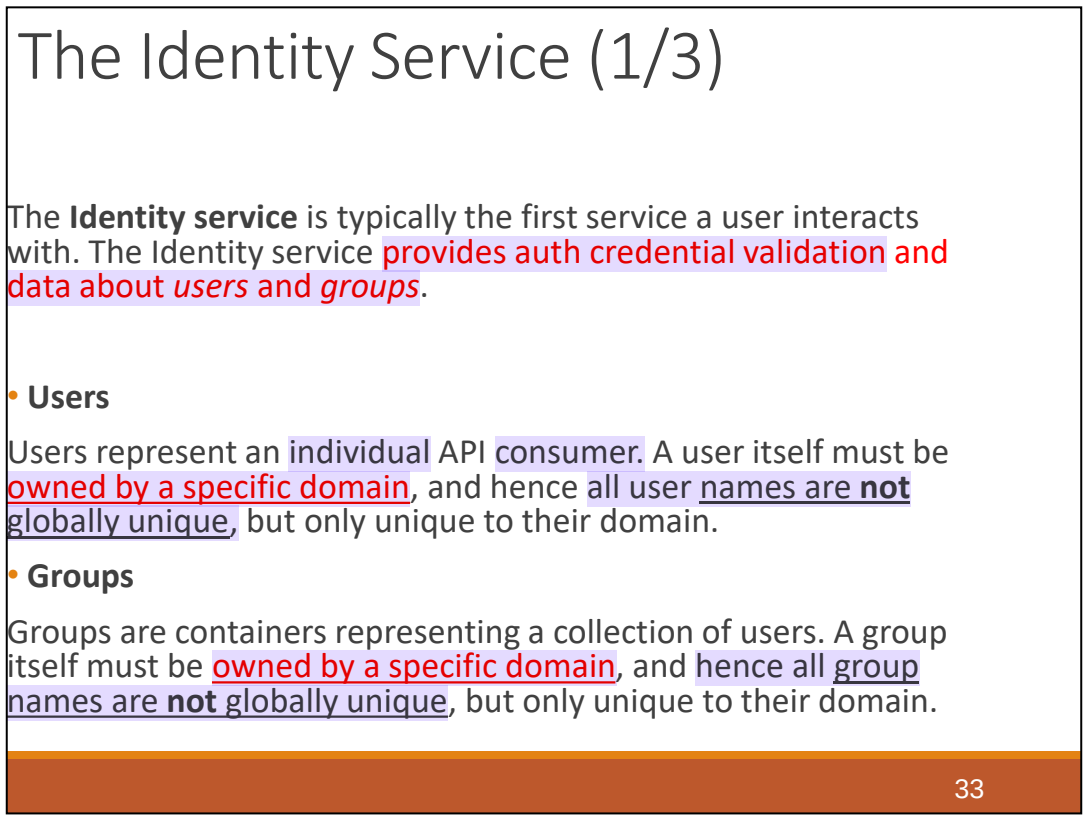
The Resource Service (2/2)

In the Identity v3 API, the uniqueness of attributes is as follows:

- **Domain Name**. Globally unique across all domains.
- **Project Name**. Unique within the owning domain.
- **User Name**. Unique within the owning domain.
- **Group Name**. Unique within the owning domain.
- **Role Name**. Unique within the owning domain.

https://docs.openstack.org/keystone/latest/admin/identity-concepts.html

in parte ripete cose della slide prima



https://docs.openstack.org/keystone/latest/admin/identity-concepts.html

Role: A personality with a defined set of user rights and privileges to perform a specific set of operations. The Identity service issues a token to a user that includes a list of roles. When a user calls a service, that service interprets the user role set, and determines to which operations or resources each role grants access.

Group: An Identity service API v3 entity. Groups are a collection of users owned by a domain. A group role, granted to a domain or project, applies to all users in the group. Adding or removing users to or from a group grants or revokes their role and authentication to the associated domain or project.

The Identity Service (2/3)

Once authenticated, an end user can use their identity to access other OpenStack resources through deployed services, based on user or group **roles**.

1. Users and services can locate other services by using the service catalog. As the name implies, a service catalog is a collection of available services and the relevant endpoints in an OpenStack deployment.

2. Each OpenStack service needs a service entry with corresponding endpoints stored in the Identity service. Each endpoint can be one of three types: **admin**, **internal**, **or public**.

34

Each service can have one or many endpoints and each endpoint can be one of three types: admin, internal, or public.

The admin API endpoint allows modifying users and tenants by default, while the public and internal APIs do not allow these operations.

In a production environment, different endpoint types might reside on separate networks exposed to different types of users for security reasons. For instance, the public API network might be visible from the Internet so customers can manage their clouds. The admin API network might be restricted to operators within the organization that manages cloud infrastructure. The internal API network might be restricted to the hosts that contain OpenStack services

The Identity Service (3/3)

The Identity service contains the following components:

- **Server**: A centralized server provides authentication and authorization services using a RESTful interface.
- **Modules**: Entities that intercept service requests, extract user credentials, and send them to the centralized server for authorization.
- **Drivers**: Components integrated to the centralized server. They are used for accessing identity information in repositories external to OpenStack (for example, SQL databases or LDAP servers).

35

The Lightweight Directory Access Protocol (LDAP) is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over an Internet Protocol (IP) network. Directory services play an important role in developing intranet and Internet applications by allowing the sharing of information about users, systems, networks, services, and applications throughout the network. As examples, directory services may provide any organized set of records, often with a hierarchical structure, such as a corporate email directory. Similarly, a telephone directory is a list of subscribers with an address and a phone number.

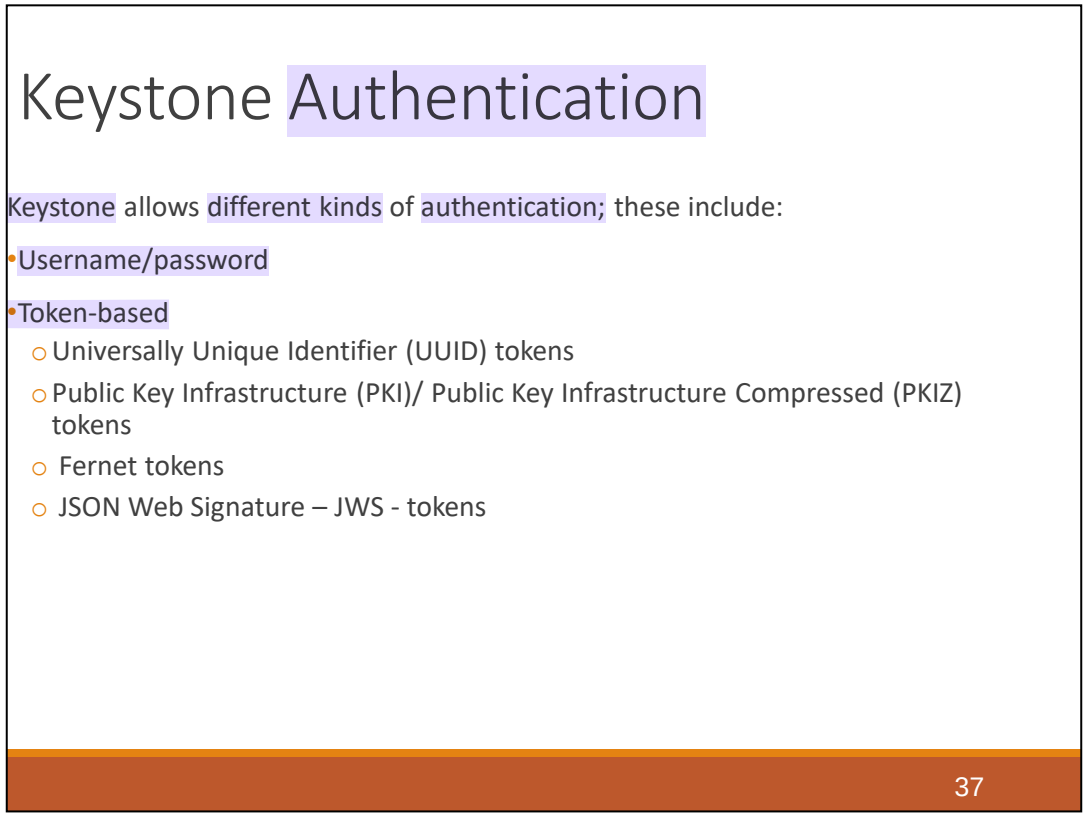
The Assignment Service (1/1)

To assign a user to a project, you must assign a role to a user-project pair. For example:

```
openstack role add --user USER_NAME --project PROJECT_NAME ROLE_NAME
```

The Assignment service provides data about *roles* and *role assignments*.

- **Roles** dictate the level of authorization the end user can obtain. Roles can be granted at either the domain or project level. A role can be assigned at the individual user or group level. Role names are unique within the owning domain.
- **Role Assignments** is therefor a 3-tuple that has a Role, a Resource and an Identity



One way to authenticate is to employ username/password. However, you would have to supply

them in every separate command to an API, which means that each command sent is a chance for

your username and password to be stolen. It also means that you have to store the username and password

locally, or you will have to re-enter them (type them in) for every separate command! It can be

dangerous to store them locally, since someone might steal them, but re-entering them every time is

just too much in a situation where you have multiple commands.

An alternative is to use tokens. A token is valid only for a limited time, which means that even if a

token is stolen when it is being sent across the network in a command to an API, it is a short time until

the thief can no longer use it. Similarly, if you store (cache) a token locally, even if it is stolen in the

case when your local account is hacked, it si a short time until the thief can no longer use it.

The Token Service

The Token service validates and manages tokens used for authenticating requests once a user's credentials have already been verified

38

The token type issued by keystone is configurable through the /etc/keystone/keystone.conf file. Currently, the only supported token provider is fernet.

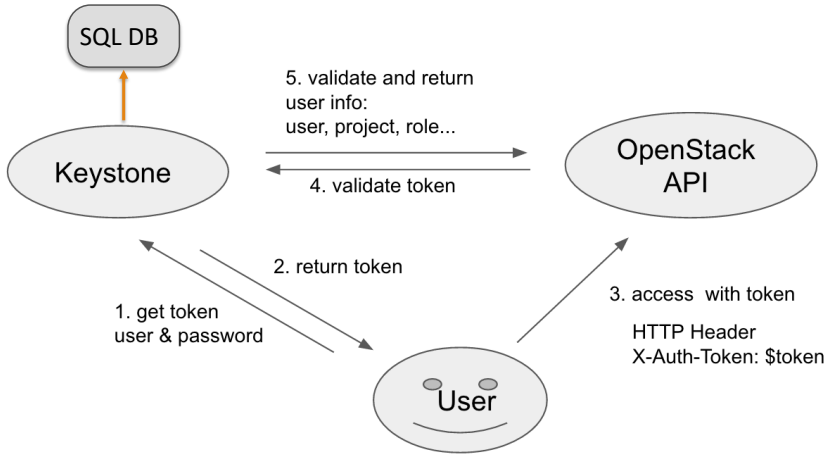
Token Types and Authentication

When configured to use **UUID tokens**, Keystone generates tokens in UUID 4 format. UUID tokens are 128 bits (16 bytes) long. e.g. 12ea1917-4641-49dc-ad68-d0dc1eb30842

The way that UUID tokens work on Keystone is as follows:

- **First stage—user acquires a token**:
  - The user sends a username/password to Keystone to request a token
  - Keystone generates a UUID token (in UUID 4 format) and sends it back to the user
  - Keystone keeps a copy of this token, together with user identification information, in its database
- **Subsequent stages—user uses the token:**
  - User sends a request to some other OpenStack component services (such as Nova or Glance) including the token
  - The OpenStack component service sends the token to Keystone for authentication
  - If the token is in the Keystone database and has not expired and has not been revoked, then Keystone tells the other OpenStack component service, okay go ahead. Otherwise Keystone tells the other OpenStack component service, nope, I don't recognize that token.

Token Types and Authentication



Authentication by UUID token

Token Types and Authentication

- The **size of tokens is a major issue**, since performance degrades when huge amounts of token data are transmitted across a network and also when huge amounts of token data are stored in a persistent token database, since authentication requests require searching for the token in persistent storage, and can take a very long time.
- This can result in the need to repeatedly flush the persistent token database. Note that some token formats are larger than others, which exacerbates the problem if used in this way.
- Keystone must periodically prune tokens from its token database, those expired, and any users whose user privileges have been revoked.

Token Types and Authentication

- One problem with the way Keystone uses UUID tokens is that every user of any OpenStack component service must send a validation message to Keystone. This can make a lot of network traffic to Keystone.
- Another important problem is the management of the token repository
- Another option is for Keystone to use **Public Key Infrastructure (PKI) tokens**.

When using PKI tokens, Keystone acts as a Certification Authority. At the time Keystone is installed, a **Certificate Authority private key**, a **Certificate Authority certificate**, a **Signing Private Key**, and a **Signing Certificate** are generated.

PKI stands for Public Key Infrastructure. Tokens are documents, cryptographically signed using the X509 standard. In order to work correctly token generation requires a public/private key pair. The public key must be signed in an X509 certificate, and the certificate used to sign it must be available as a Certificate Authority (CA) certificate. These files can be generated either using the keystone-manage utility, or externally generated. The files need to be in the locations specified by the top level Identity service configuration file /etc/keystone/keystone.conf as specified in the above section. Additionally, the private key should only be readable by the system user that will run the Identity service.

When using Public Key Infrastructure (PKI) tokens with the identity service, users must have access to the signing certificate and the certificate authority's (CA) certificate for the token issuer in order to validate tokens. This extension provides a simple means of retrieving these certificates from an identity service.

Certificates are a public resource and can be shared. Typically when validating a certificate we would only require the issuing certificate authority's certificate however PKI tokens are distributed without including the original signing certificate in the message so this must be retrievable as well.

Token Types and Authentication

When you're using either PKI or PKIZ tokens (we'll look at PKIZ tokens shortly), then the entire normal Keystone validation response is included as part of the token. This interface information in the Keystone response is called **service catalog**.


```
{
  "endpoints": [
    {
      "id": "bd130670250c4cb0bba1750a54b66be0",
      "interface": "internal",
      "region": "RegionOne",
      "region_id": "RegionOne",
      "url": "http://146.229.233.30:8773/services/Cloud"
    },
    {
      "id": "d265e0d952e2448a9b526d11358a5d9e",
      "interface": "public",
      "region": "RegionOne",
      "region_id": "RegionOne",
      "url": "http://146.229.233.30:8773/services/Cloud"
    },
```


43

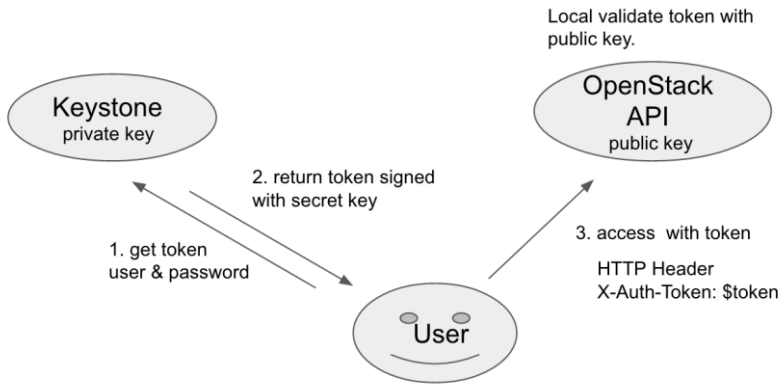
Token Types and Authentication

- This information is now converted into **Cryptographic Message Syntax (CMS)** and signed with the Certificate Authority Signing Key. The recipient will of course have to convert it back from CMS and verify the signature before processing it further.
- When a user sends a request including a PKI token to an OpenStack component service via its API, the OpenStack component service/API has a local copy of the Signing Certificate, the Certificate Authority Certificate, and the Certification Revocation List (CRL). If the OpenStack API has not previously downloaded this information from Keystone, then it does it immediately before processing the user's request.

44

The **Cryptographic Message Syntax** (**CMS**) is the IETF's standard for cryptographically protected messages. It can be used by cryptographic schemes and protocols to digitally sign, digest, authenticate or encrypt any form of digital data.

Token Types and Authentication



Authentication by PKI/PKIZ token

Token Types and Authentication

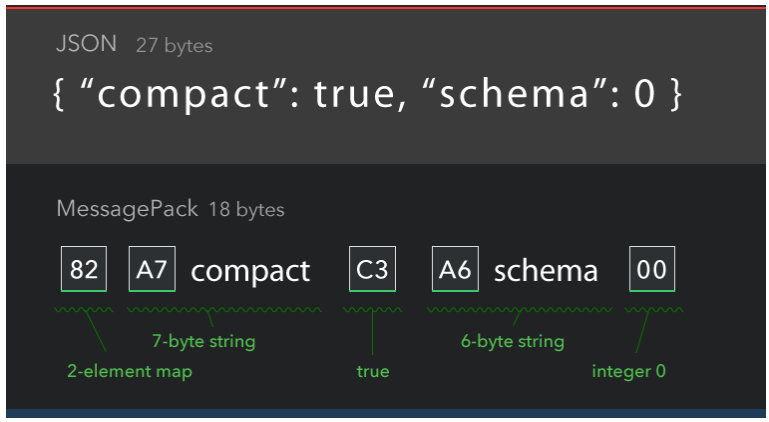
- These PKI tokens can be humongous. A basic token with a single endpoint is approximately 1700 bytes.
- The PKI token sizes increase proportionally as regions and services are added to the catalog, and sometimes can be over 8KB.
- PKIZ tokens try to fix this problem by compression— PKIZ tokens are just PKI tokens compressed using zlib compression, and except for that PKIZ works the same as PKI.

Token Types and Authentication

Fernet tokens are ephemeral, i.e. non-peristent.

- Fernet tokens contain a limited amount of identity and authorization data in a MessagePacked payload.
- MessagePack is an efficient binary serialization format. It lets you exchange data among multiple languages like JSON. But it's faster and smaller, although less readable by humans. Small integers are encoded into a single byte, and typical short strings require only one extra byte in addition to the strings themselves.

Token Types and Authentication



https://msgpack.org/

Token Types and Authentication

Fernet tokens use symmetric (secret) Key Encryption. With Fernet tokens, the data in a message (its payload) is encrypted using the Advanced Encryption Standard (AES), then the whole Fernet token is signed using a SHA256 HMAC key, and finally the whole is base 64 URL encoded. The two keys are commonly referred to as fernet key.

- Keys are held in a key repository that keystone passes to a library that handles the encryption and decryption of tokens.

49

A fernet key is used to encrypt and decrypt fernet tokens. Each key is actually composed of two smaller keys: a 128-bit AES encryption key and a 128-bit SHA256 HMAC signing key. The keys are held in a key repository that keystone passes to a library that handles the encryption and decryption of tokens.

Base64 is a group of binary-to-text encoding schemes that represent binary data (more specifically, a sequence of 8-bit bytes) in an ASCII string format by translating the data into a radix-64 representation. The term Base64 originates from a specific MIME content transfer encoding. Each non-final Base64 digit represents exactly 6 bits of data. Three 8-bit bytes (i.e., a total of 24 bits) can therefore be represented by four 6-bit Base64 digits.

Token Types and Authentication

A **key repository** is required by keystone in order to create fernet tokens. The different types are as follows:

- Primary key: There is only one primary key in a key repository. The primary key is allowed to encrypt and decrypt tokens. This key is always named as the highest index in the repository.
- Secondary key: A secondary key was at one point a primary key, but has been demoted in place of another primary key. It is only allowed to decrypt tokens. Keystone needs to be able to decrypt tokens that were created with old primary keys.
- Staged key: The staged key is a special key that shares some similarities with secondary keys. There can only ever be one staged key in a repository and it must exist. Just like secondary keys, staged keys have the ability to decrypt tokens. Unlike secondary keys, staged keys have never been a primary key. In fact, they are opposites since the staged key will always be the next primary key. This helps clarify the name because they are the next key "staged" to be the primary key. This key is always named as 0 in the key repository.

50

A fernet key is used to encrypt and decrypt fernet tokens. Each key is actually composed of two smaller keys: a 128-bit AES encryption key and a 128-bit SHA256 HMAC signing key. The keys are held in a key repository that keystone passes to a library that handles the encryption and decryption of tokens.

Token Types and Authentication

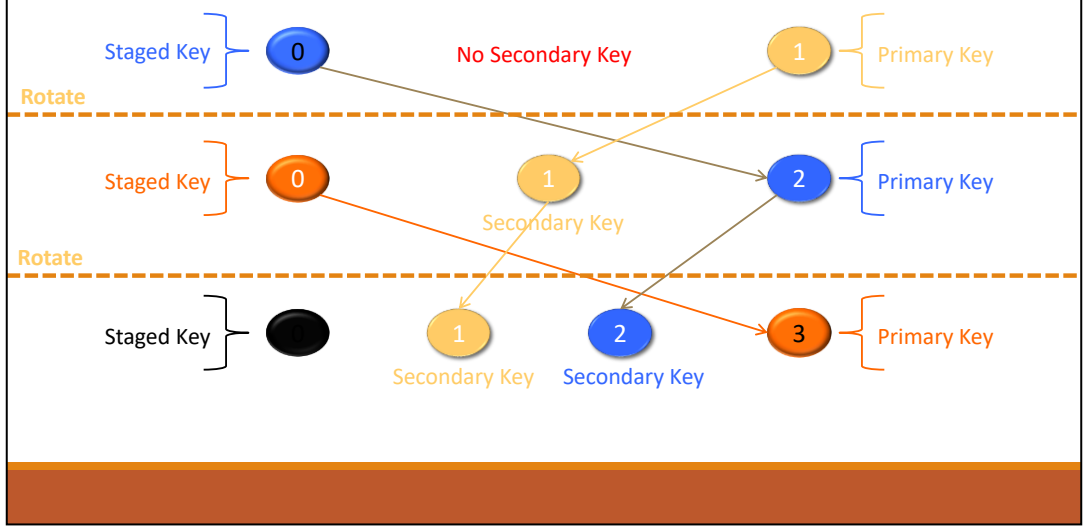
- The fernet keys have a natural lifecycle.
- Each key starts as a staged key, is promoted to be the primary key, and then demoted to be a secondary key.
- New tokens can only be encrypted with a primary key. Secondary and staged keys are never used to encrypt token.
- As an operator, this gives you the chance to perform a key rotation on one keystone node, and distribute the new key set over a span of time. This does not require the distribution to take place in an ultra short period of time
- The key repository is specified using the key_repository option in the keystone configuration file.

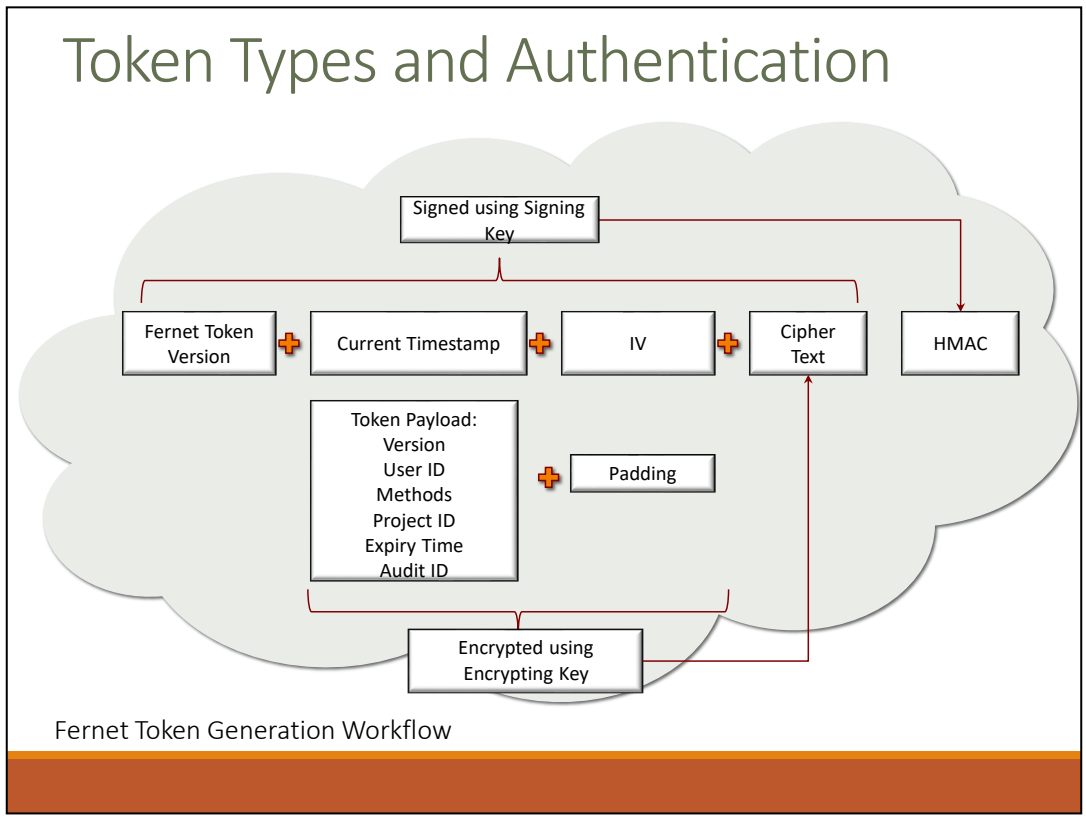
```
[fernet_tokens]
key_repository= /etc/keystone/fernet-keys/
```

ripetitivo

Token Types and Authentication

Fernet Key Rotation





Identity:

Checks if User exist in User Domain
Check if User is enabled
Retrieves **User ID**
Matches Password

Resource:

Checks if Domain or Project exist
Check if Domain or Project is enabled
Retrieves **Project ID** and **Domain ID**

Catalog:

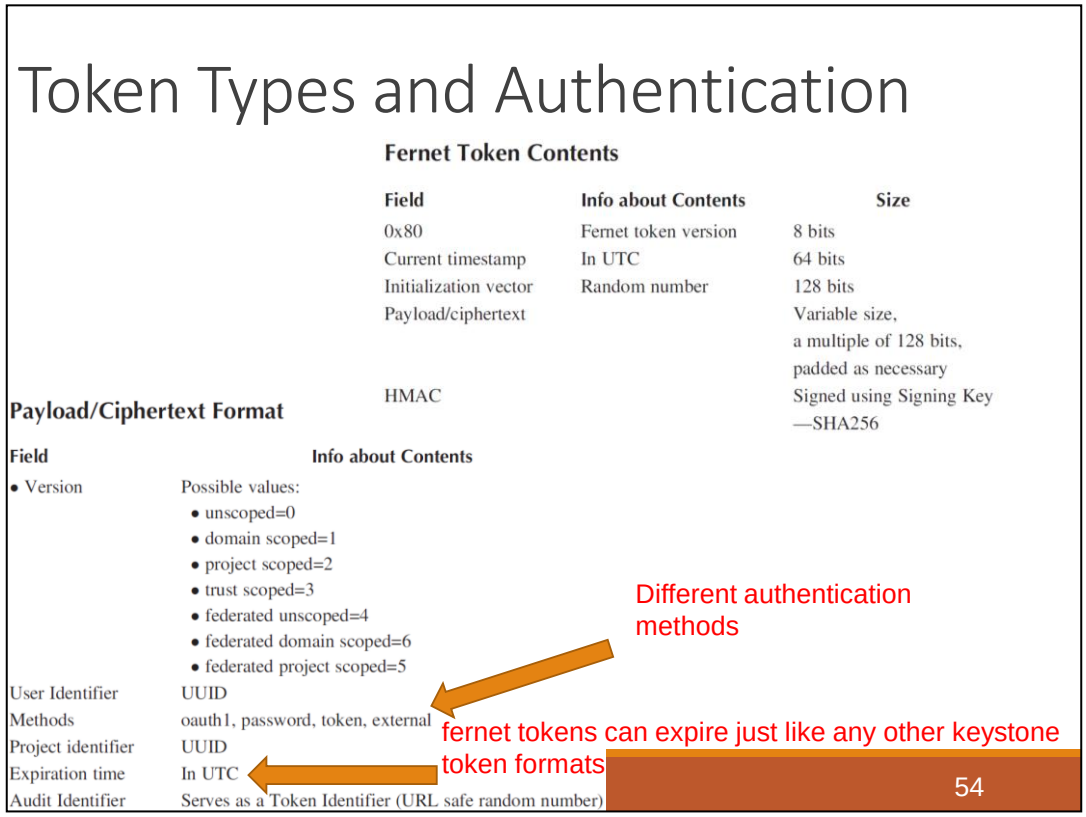
Retrieves **Services** associated with User's Project
Retrieves the list of **endpoints** for all the services

In cryptography, an HMAC (sometimes expanded as either keyed-hash message

authentication code or hash-based message authentication code) is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key. As with any MAC, it may be used to simultaneously verify both the data integrity and the authenticity of a message.

HMAC can provide message authentication using a shared secret instead of using digital signatures with asymmetric cryptography. It trades off the need for a complex public key infrastructure by delegating the key exchange to the communicating parties, who are responsible for establishing and using a trusted channel to agree on the key prior to communication

meh



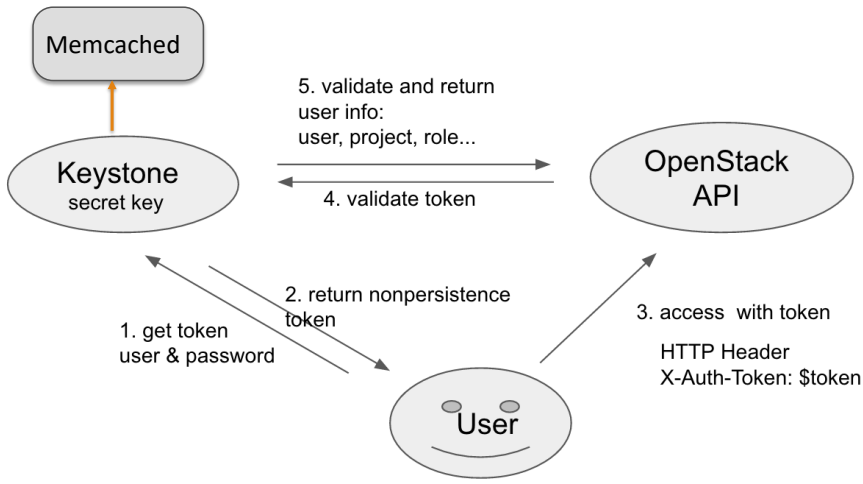
https://docs.openstack.org/keystone/latest/admin/tokens-overview.html

https://docs.openstack.org/api-ref/identity/v3-ext/#os-trust-api

A trust represents a user's (the *trustor*) authorization to delegate roles to another user (the *trustee*), and optionally allow the trustee to impersonate the trustor. After the trustor has created a trust, the trustee can specify the trust's id attribute as part of an authentication request to then create a token representing the delegated authority of the trustor.

The audit_ids attribute is a list that contains no more than two elements. Each id in the audit_ids attribute is a randomly (unique) generated string that can be used to track the token.

Token Types and Authentication



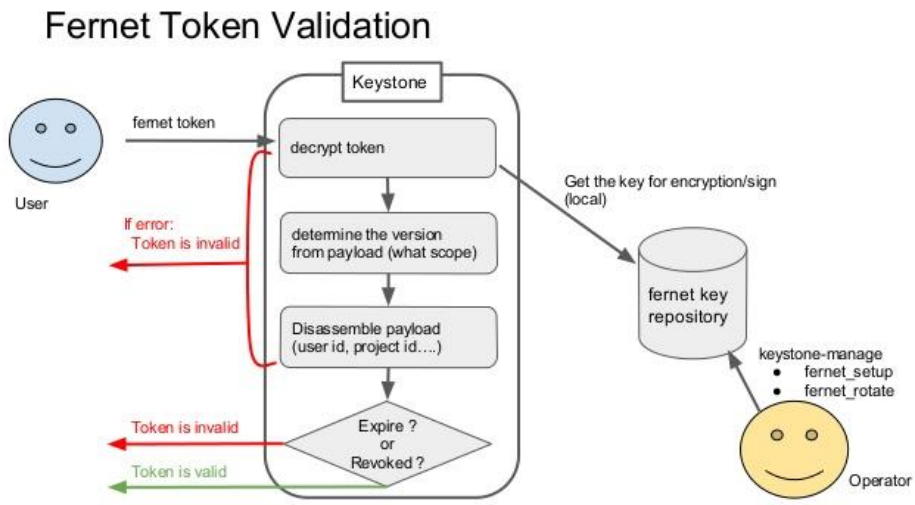
Authentication by Fernet token

55

Memcached is a open source, high-performance, distributed memory object caching system, generic in nature, but intended for use in speeding up dynamic web applications by alleviating database load. Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering.

è solo un recap di cose già dette prima

Token Types and Authentication



56

Token Types and Authentication

- Even though fernet tokens operate very similarly to UUID tokens, they do not require persistence or leverage the configured token persistence driver in any way.
  - The keystone token database no longer suffers bloat as a side effect of authentication. Pruning expired tokens from the token database is no longer required when using fernet tokens. Because fernet tokens do not require persistence, they do not have to be replicated. As long as each keystone node shares the same key repository, fernet tokens can be created and validated instantly across nodes.
- The arguments for using fernet over PKI and PKIZ remain the same as UUID, in addition to the fact that fernet tokens are much smaller than PKI and PKIZ tokens. PKI and PKIZ tokens still require persistent storage and can sometimes cause issues due to their size. Fernet tokens are kept under a 250 byte limit. PKI and PKIZ tokens typically exceed 1600 bytes in length.

JSON Web Signature (JWS) token

- Implemented in the Stein release. **JWS tokens are signed**, meaning the information used to build the token ID is not opaque to users and can it can be decoded by anyone.
- JWS tokens are ephemeral, or non-persistent, which means they won't bloat the database or require replication across nodes.
- Tokens are signed with private keys and validated with public keys. The JWS token provider implementation only supports the ES256 JSON Web Algorithm (JWA), which is an Elliptic Curve Digital Signature Algorithm (ECDSA) using the P-256 curve and a SHA-256 hash algorithm.

JSON web token (JWT), pronounced "jot", is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. Because of its relatively small size, a JWT can be sent through a URL, through a POST parameter, or inside an HTTP header, and it is transmitted quickly. A JWT contains all the required information about an entity to avoid querying a database more than once. The recipient of a JWT also does not need to call a server to validate the token.

meh

JSON Web Signature (JWS) token

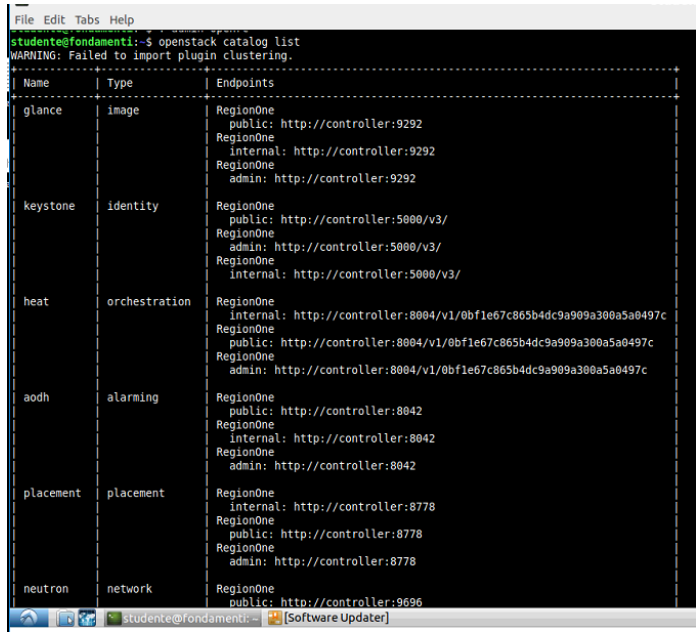
- A deployment might consider using JWS tokens as **opposed to fernet tokens** if there are security concerns about key sharing. Remember that key distribution is only required in multi-node keystone deployments. If you only have one keystone node serving requests in your deployment, key distribution is unnecessary.
- Note that a major difference between the two providers is that JWS tokens are not opaque and can be decoded by anyone with the token ID. Fernet tokens are opaque in that the token ID is ciphertext.
- Despite the JWS token payload being readable by anyone, keystone reserves the right to make backwards incompatible changes to the token payload itself, which is not an API contract. It is recommended validating the token against keystone's authentication API to inspect its associated metadata.

A deployment might consider using JWS tokens as opposed to fernet tokens if there are security concerns about sharing symmetric encryption keys across hosts. Note that a major difference between the two providers is that JWS tokens are not opaque and can be decoded by anyone with the token ID. Fernet tokens are opaque in that the token ID is ciphertext. Despite the JWS token payload being readable by anyone, keystone reserves the right to make backwards incompatible changes to the token payload itself, which is not an API contract. We only recommend validating the token against keystone's authentication API to inspect its associated metadata.

The Catalog Service (1/2)

- The OpenStack keystone service **catalog** allows API clients to dynamically discover and navigate through cloud services.
- The service catalog may differ from deployment-to-deployment, user-to-user, and project-to-project.
- The Catalog service provides an endpoint registry used for endpoint discovery

The Catalog Service (2/2)



The Policy Service

- The Policy service provides a rule-based authorization engine and the associated rule management interface.
- Each OpenStack service defines the access policies for its resources in an associated policy file. A resource, for example, could be API access, the ability to attach to a volume, or to fire up instances. The policy rules are specified in JSON format and the file is called policy.json.

62

Each OpenStack service, Identity, Compute, Networking, and so on, has its own role-based access policies. They determine which user can access which objects in which way, and are defined in the service's policy.json file.

Whenever an API call to an OpenStack service is made, the service's policy engine uses the appropriate policy definitions to determine if the call can be accepted. Any changes to policy.json are effective immediately, which allows new policies to be implemented while the service is running.

A policy.json file is a text file in JSON (Javascript Object Notation) format. Each policy is defined by a one-line statement in the form "<target>" : "<rule>".

The Policy Service

/usr/lib/python2.7/dist-packages/cinder/tests/unit/policy.json

/usr/lib/python2.7/dist-packages/glance/tests/etc/policy.json

/usr/lib/python2.7/dist-packages/gnocchi/rest/policy.json

/usr/lib/python2.7/dist-packages/neutron/tests/etc/policy.json

/usr/lib/python2.7/dist-packages/neutron_lib/tests/etc/no_policy.json

/usr/lib/python2.7/dist-packages/neutron_lib/tests/etc/policy.json

/usr/lib/python2.7/dist-packages/openstack_auth/tests/conf/keystone_policy.json

/usr/lib/python2.7/dist-packages/openstack_auth/tests/conf/no_default_policy.json

/usr/lib/python2.7/dist-packages/openstack_auth/tests/conf/nova_policy.json

/usr/lib/python2.7/dist-packages/openstack_auth/tests/conf/with_default_policy.json

/usr/lib/python2.7/dist-packages/openstack_dashboard/conf/cinder_policy.json

/usr/lib/python2.7/dist-packages/openstack_dashboard/conf/glance_policy.json

/usr/lib/python2.7/dist-packages/openstack_dashboard/conf/keystone_policy.json

/usr/lib/python2.7/dist-packages/openstack_dashboard/conf/neutron_policy.json

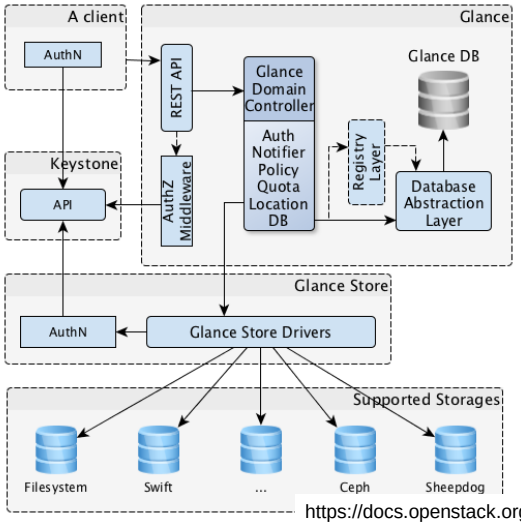
/usr/lib/python2.7/dist-packages/openstack_dashboard/conf/nova_policy.json

Try:

$ locate policy.json

OpenStack Glance: Image Service

Glance is an OpenStack project that provides a service where users can upload and discover data assets that are meant to be used with other services. This currently includes **images and metadata** definitions. It implements OpenStack's **Images API v2**



- **Core Use Cases:**
  - Glance provisions and manages images of Virtual Machines (VMs).
- **Key Capabilities:**
  - Glance image services include discovering, registering, and retrieving virtual machine (VM) images.
  - Glance has a RESTful API that allows querying of VM image metadata as well as retrieval of the actual image.

https://docs.openstack.org/glance/pike/contributor/architecture.html

OpenStack Glance has a client-server architecture that provides the user with a REST A
A **Glance Domain Controller** manages the internal server operations that is divided into
All the file (Image data) operations are performed using **glance_store** library, which is re
Glance uses a central database (**Glance DB**) that is shared amongst all the components

Following components are present in the Glance architecture:
A client - any application that makes use of a Glance server.
REST API - Glance functionalities are exposed via REST.
Database Abstraction Layer (DAL) - an application programming interface (API) that u
Glance Domain Controller - middleware that implements the main Glance functionalitie
Glance Store - used to organize interactions between Glance and various data stores.
Registry Layer - optional layer that is used to organise secure communication between

Authn (authentication) primarily deals with user identity: who is this person? Is she wh
Authz (authorization) performs the authentication token validation and retrieves actual

The **domain model** contains the following layers:

Authorization

Property protection

Notifier

Policy

Quota

Location

Database

Authorization¶

The first layer of the domain model provides a verification of whether an image itself or its property can be changed. An admin or image owner can apply the changes. The information about a user is taken from the request context and is compared with the image owner. If the user cannot apply a change, a corresponding error message appears.

Property protection¶

The second layer of the domain model is optional. It becomes available if you set the property_protection_file parameter in the Glance configuration file.

There are two types of image properties in Glance:

Core properties, as specified in the image schema

Meta properties, which are the arbitrary key/value pairs that can be added to an image

The property protection layer manages access to the meta properties through Glance's public API calls. You can restrict the access in the property protection configuration file.

Notifier¶

On the third layer of the domain model, the following items are added to the message queue:

Notifications about all of the image changes

All of the exceptions and warnings that occurred while using an image

Policy¶

The fourth layer of the domain model is responsible for:

Defining access rules to perform actions with an image. The rules are defined in the etc/policy.yaml file.

Monitoring of the rules implementation.

Quota¶

On the fifth layer of the domain model, if a user has an admin-defined size quota for all of his uploaded images, there is a check that verifies whether this quota

exceeds the limit during an image upload and save:

If the quota does not exceed the limit, then the action to add an image succeeds.

If the quota exceeds the limit, then the action does not succeed and a corresponding error message appears.

Location¶

The sixth layer of the domain model is used for interaction with the store via the glance_store library, like upload and download, and for managing an image location. On this layer, an image is validated before the upload. If the validation succeeds, an image is written to the glance_store library.

This sixth layer of the domain model is responsible for:

Checking whether a location URI is correct when a new location is added

Removing image data from the store when an image location is changed

Preventing image location duplicates

Database¶

On the seventh layer of the domain model:

The methods to interact with the database API are implemented.

Images are converted to the corresponding format to be recorded in the database. And the information received from the database is converted to an Image object.

Glance Components

- **A client** - any application that makes use of a Glance server.
- **REST API** - Glance capabilities are exposed via REST.
- **Database Abstraction Layer (DAL)** - an application programming interface (API) that unifies the communication between Glance and databases.
- **Glance Domain Controller** - middleware that implements the main Glance functions such as authorization, notifications, policies, database connections.
- **Glance Store** - used to organize interactions between Glance and various data stores.

Glance Components

- **Registry Layer** - optional layer that is used to organise secure communication between the domain and the DAL by using a separate service.

- **Database**: Stores image metadata and you can choose your database depending on your preference. Most deployments use MySQL or SQLite.

Glance General Features

- VM images made available through Glance can be stored in a variety of locations from simple filesystems to object-storage systems like the OpenStack Swift project.
- Images are uniquely identified by way of a URI that matches the following signature:
  - <Glance Server Location>/v2/images/<ID>
  - /var/lib/glance/images in case of filesystem backend.
- When you add an image to the Image service, you can specify its **disk** and **container** formats.
- You can set your image's disk format to one of the following:
  - **raw, vhd, vhdx, vmdk, vdi, iso, ploop, qcow2, aki, ari, ami.**

67

The raw image format is the simplest one, and is natively supported by both KVM and Xen hypervisors. You can think of a raw image as being the bit-equivalent of a block device file, created as if somebody had copied, say, /dev/sda to a file using the **dd** command.

You can set your image's disk format to one of the following:

raw

This is an unstructured disk image format

vhd

This is the VHD disk format, a common disk format used by virtual machine monitors from VMware, Xen, Microsoft, VirtualBox, and others

vhdx

This is the VHDX disk format, an enhanced version of the vhd format which supports larger disk sizes among other features.

vmdk

Another common disk format supported by many common virtual machine monitors

vdi

A disk format supported by VirtualBox virtual machine monitor and the QEMU emulator

iso

An archive format for the data contents of an optical disc (e.g. CDROM).

ploop

A disk format supported and used by Virtuozzo to run OS Containers

qcow2

A disk format supported by the QEMU emulator that can expand dynamically and supports Copy on Write

aki

This indicates what is stored in Glance is an Amazon kernel image

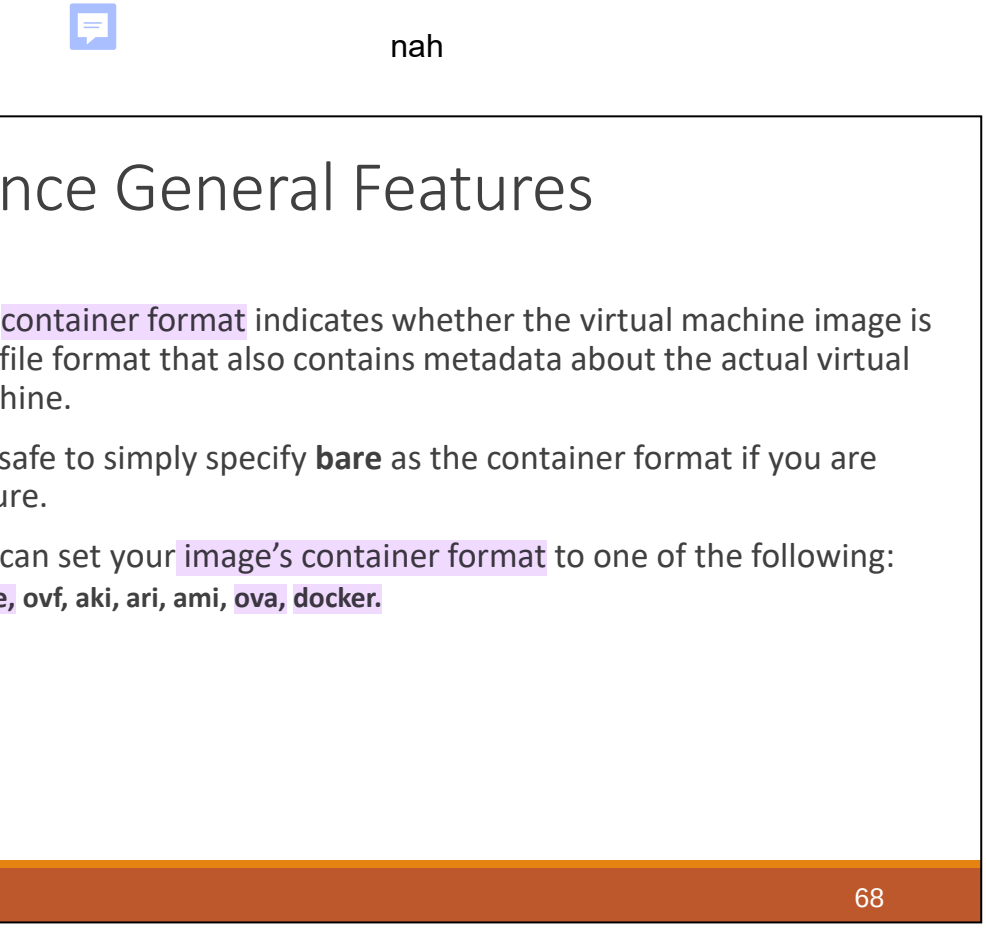
ari

This indicates what is stored in Glance is an Amazon ramdisk image

ami

This indicates what is stored in Glance is an Amazon machine image

nah



The container format refers to whether the virtual machine image is in a file format that also contains metadata about the actual virtual machine.

You can set your image's container format to one of the following:

bare

This indicates there is no container or metadata envelope for the image

ovf

This is the OVF container format. **Open Virtualization Format** (**OVF**) is an open standard for packaging and distributing virtual appliances or, more generally, software to be run in virtual machines.

aki

This indicates what is stored in Glance is an Amazon kernel image

ari

This indicates what is stored in Glance is an Amazon ramdisk image

ami

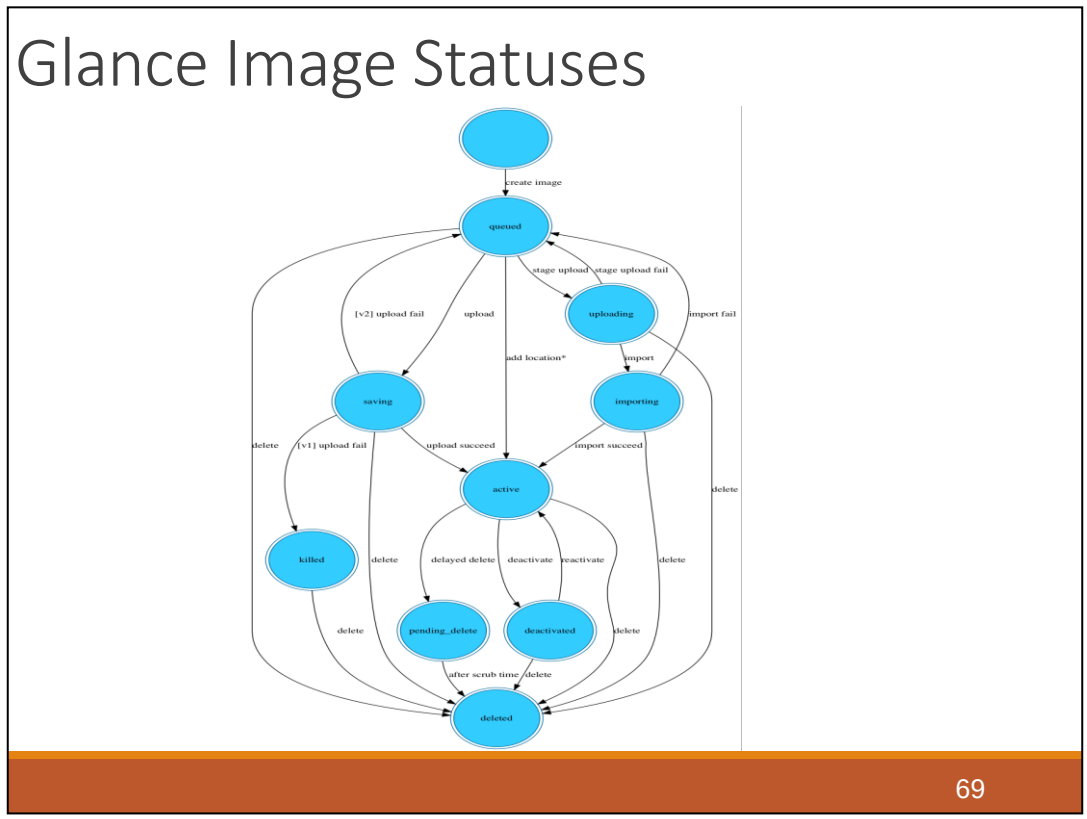
This indicates what is stored in Glance is an Amazon machine image

ova

This indicates what is stored in Glance is an OVA tar archive file

docker

This indicates what is stored in Glance is a Docker tar archive of the container filesystem



queued: The image identifier has been reserved for an image in the Glance registry. No image data has been uploaded to Glance and the image size was not explicitly set to zero on creation.

saving: Denotes that an image's raw data is currently being uploaded to Glance. When an image is registered with a call to POST /images and there is an x-image-meta-location header present, that image will never be in the saving status (as the image data is already available in some other location).

uploading: Denotes that an import data-put call has been made. While in this status, a call to PUT /file is disallowed. (Note that a call to PUT /file on a queued image puts the image into saving status. Calls to PUT /stage are disallowed while an image is in saving status. Thus it's not possible to use both upload methods on the same image.)

importing: Denotes that an import call has been made but that the image is not yet ready for use.

active: Denotes an image that is fully available in Glance. This occurs when the image data is uploaded, or the image size is explicitly set to zero on creation.

deactivated: Denotes that access to image data is not allowed to any non-admin user. Prohibiting downloads of an image also prohibits operations like image export and image cloning that may require image data.

killed: Denotes that an error occurred during the uploading of an image's data, and that the image is not readable.

deleted: Glance has retained the information about the image, but it is no

longer available to use. An image in this state will be removed automatically at a later date.

pending_delete: This is similar to deleted, however, Glance has not yet removed the image data. An image in this state is not recoverable.

Glance Image Statuses

Status	Description
queued	The Image service reserved an image ID for the image in the catalog but did not yet upload any image data.
saving	The Image service is in the process of saving the raw data for the image into the backing store.
active	The image is active and ready for consumption in the Image service.
killed	An image data upload error occurred.
deleted	The Image service retains information about the image but the image is no longer available for use.
pending_delete	Similar to the deleted status. An image in this state is not recoverable.
deactivated	The image data is not available for use.
uploading	Data has been staged as part of the interoperable image import process. It is not yet available for use. *(Since Image API 2.6)*
importing	The image data is being processed as part of the interoperable image import process, but is not yet available for use. *(Since Image API 2.6)*

Glance Image Visibility

Visibility	Description
public	**Any user** may read the image and its data payload. Additionally, the image appears in the **default image list of all users.**
community	**Any user** may read the image and its data payload, but the image **does *not*** appear in the **default image list of any user** other than the owner**.**
shared	The image is associated with an ***image members list.*** **Only the owner and users in the image members list may read the image or its data payload.** The image appears in the **default image list of the owner.** It also appears in the **default image list of members who have *accepted* to be in the image member list.** If you do not specify a visibility value when you create an image, it is assigned this visibility by default. **Non-owners, however, will not have access to the image** until they are added as image member list**.**
private	**Only the owner** image may read the image or its data payload. Additionally, the image appears in the **owner's default image list.**

Visibility Semantics

Here are the semantics for visibility:

1.visibility is orthogonal to ownership

2.semantics for each of the values

1. **public**: all users:
    1. have this image in default image-list
    2. can see image-detail for this image
    3. can boot from this image
2. **private**: users with tenantId == tenantId(owner) *only*:
    1. have this image in the default image-list
    2. see image-detail for this image
    3. can boot from this image
3. **shared**:
    1. users with tenantId == tenantId(owner)
        1. have this image in the default image-list
        2. see image-detail for this image
        3. can boot from this image
    2. users with tenantId in the member-list of the image
        1. can see image-detail for this image

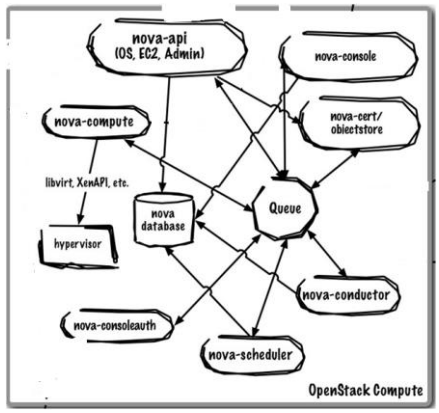
        2. can boot from this image
    1. users with tenantId in the member-list with member_status == 'accepted'
        1. have this image in their default image-list

1. **community**:
    1. all users
        1. can see image-detail for this image
        2. can boot from this image
    2. users with tenantId in the member-list of the image with member_status == 'accepted'
        1. have this image in their default image-list

1.NOTE: it's possible for an image to have 'visibility' == 'shared' but have an empty member-list

OpenStack Nova

Nova is an OpenStack project that provides users with computing resources. It implements OpenStack's **Compute API**



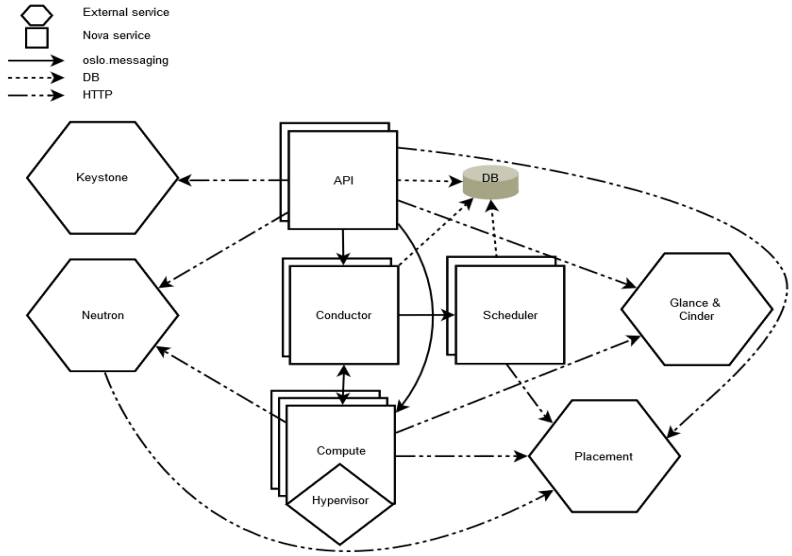
- **Core Use Cases:**

Nova provisions and manages Virtual Machines (VMs).

- **Key Capabilities:**

OpenStack Nova can work with several different hypervisors.

Nova Components



https://docs.openstack.org/nova/latest/install/get-started-compute.html

https://docs.openstack.org/placement/latest/

The placement API service was introduced in the 14.0.0 Newton release within the nova repository and extracted to the placement repository in the 19.0.0 Stein release. This is a REST API stack and data model used to track resource provider inventories and usages, along with different classes of resources. For example, a resource provider can be a compute node, a shared storage pool, or an IP allocation pool. The placement service tracks the inventory and usage of each provider. For example, an instance created on a compute node may be a consumer of resources such as RAM and CPU from a compute node resource provider, disk from an external shared storage pool resource provider and IP addresses from an external IP pool resource provider.

Two processes, nova-compute and nova-scheduler, host most of nova's interaction with placement.

Nova Components

- DB: sql database for data storage.
- API: component that receives HTTP requests, converts commands and communicates with other components via the **oslo.messaging** queue or HTTP.
- Scheduler: decides which host gets each instance.
- Compute: manages communication with hypervisor and virtual machines.
- Conductor: handles requests that need coordination (build/resize), acts as a database proxy, or handles object conversions.
- Placement: tracks resource provider inventories and usages (external service).

Nova Components

- Nova **SQL database**: Stores most build-time and run-time states for a cloud infrastructure, including:
  - Available instance types
  - Instances in use
  - Available networks
  - Projects
    - Theoretically, OpenStack Compute can support any database that SQLAlchemy supports. Common databases are SQLite3 for test and development work, **MySQL**, **MariaDB**, and **PostgreSQL**.
- **nova-compute service**: A worker daemon that creates and terminates virtual machine instances through hypervisor APIs, for example for Xen, QEMU, Vmware.
- **nova-scheduler service**: Takes a virtual machine instance request from the queue and determines on which compute server host it runs.

meh

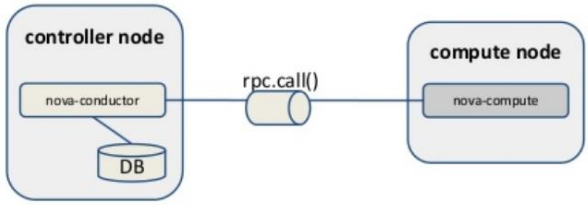
Nova Components

Nova consists of several components, which can typically run on different servers:

- **Message Queue**—central hub for passing messages between components. Uses Advanced Message Queuing Protocol (AMQP). By default uses **RabbitMQ**, can use Apache Qpid
- **Nova API**—handles OpenStack RESTful API, EC2, and admin interfaces.
  - **nova-api service**: Accepts and responds to end user compute API calls. Provides an endpoint, initiates running and instances, enforces some policies.
  - **nova-api-metadata** service: Accepts metadata requests from instances.
- **nova-conductor** — Mediates interactions between the nova-compute service and the database. It eliminates direct accesses to the cloud database made by the nova-compute service. Conceptually, it implements a new layer on top of nova-compute. The nova-conductor module scales horizontally.

Nova Components

Nova-conductor – compute exchange



Nova components use Remote Procedure Calls to communicate to one another. This paradigm is built atop the publish/subscribe paradigm implemented by RabbitMQ

- Stateless RPC server. Acts as a proxy to the database.
- Eliminates remote DB access (security)
- Horizontal scalability; spawn multiple worker threads operating in parallel (performance)
- Hides DB implementation from the Nova Compute (upgrades)
- Beneficial for operations that cross multiple compute nodes (migration, resizes).

77

The nova-conductor service enables OpenStack to function without compute nodes accessing the database. Conceptually, it implements a new layer on top of nova-compute. It should not be deployed on compute nodes, or else the security benefits of removing database access from nova-compute are negated. Just like other nova services such as nova-api or nova-scheduler, it can be scaled horizontally. You can run multiple instances of nova-conductor on different machines as needed for scaling purposes.

The Nova conductor does not expose a REST API, but communicates with the other Nova components via RPC calls (based on RabbitMQ). The conductor is used to handle long-running tasks like building an instance or performing a live migration.

From https://docs.openstack.org/nova/queens/reference/rpc.html

Nova components (the compute fabric of OpenStack) use Remote Procedure Calls (RPC hereinafter) to communicate to one another; however such a paradigm is built atop the publish/subscribe paradigm so that the following benefits can be achieved:

•Decoupling between client and servant (such as the client does not need to know where the servant's reference is).

•Full a-synchronism between client and servant (such as the client does not need

the servant to run at the same time of the remote call).

•Random balancing of remote calls (such as if more servants are up and running, one-way calls are transparently dispatched to the first available servant).

- RabbitMQ broker stays between internal components of each service in OpenStack (e.g. nova-conductor, nova-scheduler in Nova service) and allow them to communicate each other in the loosely coupled way.

- RabbitMQ runs on controller as a process that will get the message from Invoker (API or scheduler) through rpc.call() or rpc.cast().

Nova Message Exchange

OpenStack communicates between internal components using a **Message Queue** and **Advanced Message Queuing Protocol (AMQP)**. This uses a **publish/subscribe** message paradigm Message-Oriented Middleware.

- Messages can be distributed from one producer to many consumers, or from many producers to many consumers. For example, all consumers that subscribe to a particular topic (channel) will receive the message.
- An AMQP message broker handles communication between any two Nova components and messages are queued.
- This means that the Nova components are only loosely coupled.

Placement Service

- The placement API service was introduced in the 14.0.0 Newton release within the nova repository and extracted to the placement repository in the 19.0.0 Stein release.
- In a fully installed system, OpenStack services can register as **resource providers**. Each provider offers a certain set of resource classes, which is called an **inventory**.
- This is a REST API stack and data model used to track resource provider **inventories** and **usages**, along with different classes of resources.
  - For example, a resource provider can be a compute node, a shared storage pool, or an IP allocation pool.
- **The placement service tracks the inventory and usage of each provider**. For example, an instance created on a compute node may be a consumer of resources such as RAM and CPU from a compute node resource provider, disk from an external shared storage pool resource provider and IP addresses from an external IP pool resource provider.
- To link consumers and usage information, Placement uses **allocations**. An allocation represents a usage of resources by a specific consumer,

Before the Stein release the placement code was in Nova alongside the compute REST API code (nova-api). Now the placement API is a separate service and thus should be registered under a *placement* service type in the service catalog. Clients of placement, such as the resource tracker in the nova-compute node, will use the service catalog to find the placement endpoint.

The placement service provides an HTTP API <https://docs.openstack.org/api-ref/placement/> used to track resource provider inventories and usages, along with different classes of resources. For example, a resource provider can be a compute node, a shared storage pool, or an IP allocation pool.

A list of all *resource classes is* known to Placement. Resource classes are types of resources that Placement manages, like IP addresses, vCPUs, disk space or memory. In a fully installed system, OpenStack services can register as *resource providers*. Each provider offers a certain set of resource classes, which is called an *inventory*. A compute node, for instance, would typically provide CPUs, disk space and memory.

For each resource provider, Placement also maintains *usage data* which keeps track of the current usage of the resources.

To link *consumers* and usage information, Placement uses *allocations*. An allocation represents a usage of resources by a specific consumer,

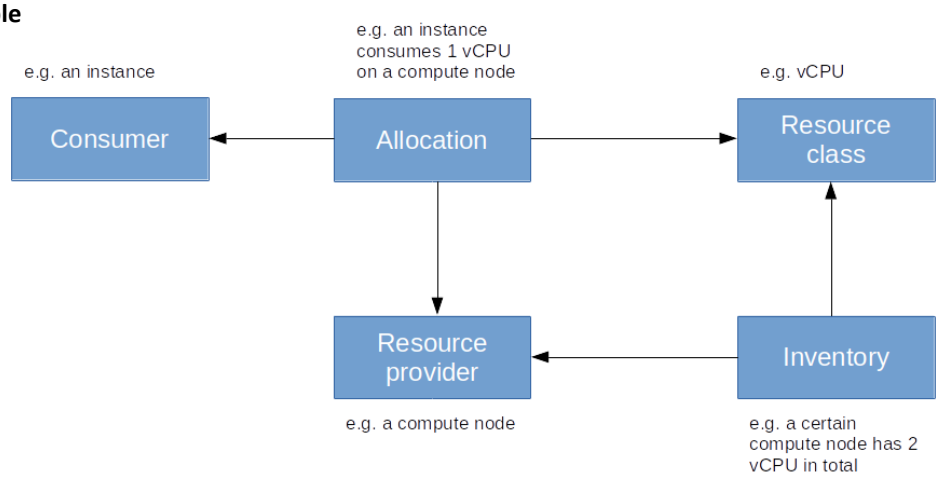
Relation between Nova and Placement. As we have mentioned above, compute nodes represent resource providers in Placement, so Nova needs to register resource provider records for the compute nodes it manages and providing the inventory information. When a new instance is created, the Nova scheduler will request information on inventories and current usage from Placement to determine the compute node on which the instance will be placed, and will subsequently update the allocations to register itself as a consumer for the resources consumed by the newly created

instance.

meh

Placement Service

Example



80

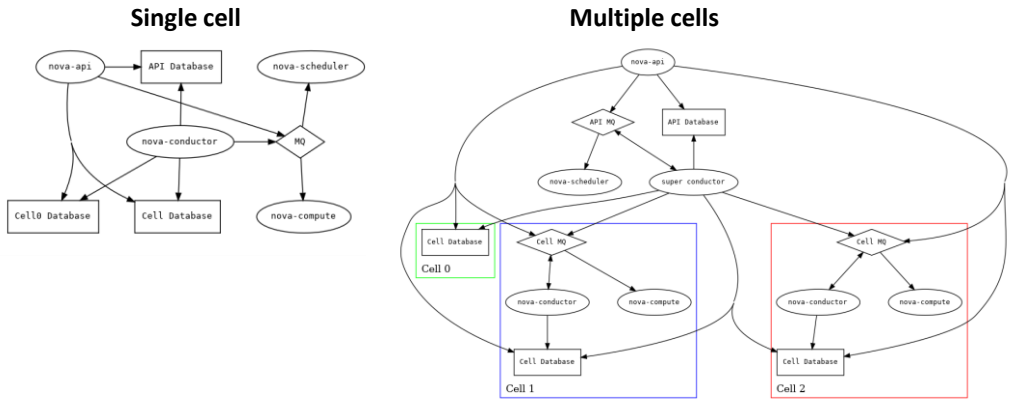
OpenStack Cells

- Historically, Nova has depended on a single logical database and message queue that all nodes depend on for communication and data persistence. This becomes an issue for deployers as scaling and providing fault tolerance for these systems is difficult.
- An experimental feature in Nova called “cells”, is used by some large deployments to partition compute nodes into smaller groups, coupled with a database and queue.
- When this functionality is enabled, the hosts in an OpenStack Compute cloud are partitioned into groups called cells. Cells are configured as a tree.
- The purpose of the cells functionality in nova is to allow larger deployments to **shard** their many compute nodes into cells. All nova deployments are by definition cells deployments, even if most will only ever have a single cell. This means a multi-cell deployment will not be radically different from a “standard” nova deployment.

The idea behind this is that the set of your compute nodes are partitioned into cells. Every compute node is part of a cell, and in addition to these regular cells, there is a cell called cell0 (which is usually not used and only holds instances which could not be scheduled to a node). The Nova database schema is split into a global part which is stored in a database called the API database and a cell-local part. This cell-local database is different for each cell, so each cell can use a different database running (potentially) on a different host. A similar sharding applies to message queues. When you set up a compute node, the configuration of the database connection and the connection to the RabbitMQ service determine to which cell the node belongs. The compute node will then use this database connection to register itself with the corresponding cell database, and a special script (nova-manage) needs to be run to make these hosts visible in the API database as well so that they can be used by the scheduler.

OpenStack Cells

The top-level cell should have a host that runs a nova-api service, but no nova-compute services. Each child cell should run all of the typical nova-* services in a regular Compute cloud except for nova-api. You can think of cells as a normal Compute deployment in that each cell has its own database server and message queue broker.



https://docs.openstack.org/nova/latest/admin/cells.html

MQ: Message Queue

A “cell database” which is used by API, conductor and compute services, and which houses the majority of the information about instances.

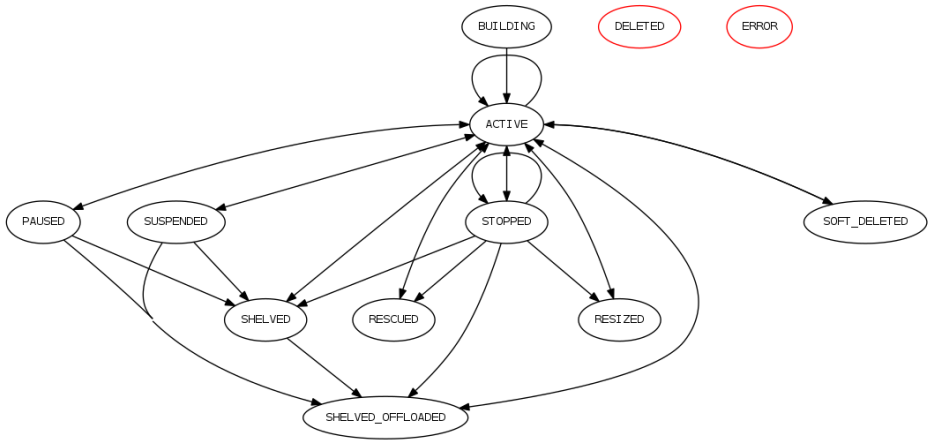
A “cell0 database” which is just like the cell database, but contains only instances that failed to be scheduled. This database mimics a regular cell, but has no compute nodes and is used only as a place to put instances that fail to land on a real compute node (and thus a real cell).

A message queue which allows the services to communicate with each other via RPC.

In larger deployments, we can opt to shard the deployment using multiple cells. In this configuration there will still only be one global API database but there will be a cell database (where the bulk of the instance information lives) for each cell, each containing a portion of the instances for the entire deployment within, as well as per-cell message queues and per-cell nova-conductor instances. There will also be an additional nova-conductor instance, known as a super conductor, to handle API-level operations.

NON VANNO SAPUTI A MEMORIA, solo il concetto generale di questi stati transitori tra ACTIVE & ERROR

Virtual Machine States and Transitions



All states are allowed to transition to DELETED and ERROR.

https://docs.openstack.org/nova/rocky/reference/vm-states.html

https://docs.openstack.org/nova/rocky/reference/vm-states.html

Virtual Machine (Server) States and Transitions

- **INITIALIZED**: VM is just created in the database, but has not been built. (was BUILDING)
- **ACTIVE**: VM is running with the specified image.
- **RESCUED**: VM is running with the rescue image.
- **PAUSED**: VM is paused with the specified image.
- **SUSPENDED**: VM is suspended with the specified image, with a valid memory snapshot.
- **STOPPED**: VM is not running, and the image is on disk.
- **SOFT_DELETED**: VM is no longer running on compute, but the disk image remains and can be brought back. The server is marked as deleted but will remain in the cloud for some configurable amount of time. While soft-deleted, an authorized user can restore the server back to normal state. When the time expires, the server will be deleted permanently
- **HARD_DELETED**: From quota and billing's perspective, the VM no longer exists. VM will eventually be destroyed running on compute, disk images too.
- **RESIZED**: The VM is stopped on the source node but running on the destination node. The VM images exist at two locations (src and dest, with different sizes). The user is expected to confirm the resize or revert it.

https://docs.openstack.org/nova/rocky/reference/vm-states.html

Virtual Machine (Server) States and Transitions

- **ERROR**: some unrecoverable error happened. Only delete is allowed to be called on the VM.
- **SHELVED:** The server is in shelved state. Depends on the shelve offload time, the server will be automatically shelved off loaded.
- **SHELVED_OFFLOADED**: The shelved server is offloaded (removed from the compute host) and it needs unshelved action to be used again.

https://docs.openstack.org/nova/rocky/reference/vm-states.html

Shelving is useful if you have an instance that you are not using, but would like retain in your list of servers. For example, you can stop an instance at the end of a work week, and resume work again at the start of the next week. All associated data and resources are kept; however, anything still in memory is not retained. If a shelved instance is no longer needed, it can also be entirely removed.

You can run the following shelving tasks:

•Shelve an instance - Shuts down the instance, and stores it together with associated data and resources (a snapshot is taken if not volume backed). Anything in memory is lost.

$ nova shelve SERVERNAME

Note By default, the **nova shelve** command gives the guest operating system a chance to perform a controlled shutdown before the instance is powered off. The shutdown behavior is configured by the shutdown_timeout parameter that can be set in the nova.conf file. Its value stands for the overall period (in seconds) a guest operation system is allowed to complete the shutdown. The default timeout is 60 seconds. See Description of Compute configuration options for details.

The timeout value can be overridden on a per image basis by means of

os_shutdown_timeout that is an image metadata setting allowing different types of operating systems to specify how much time they need to shut down cleanly.

•Unshelve an instance - Restores the instance.

$ nova unshelve SERVERNAME

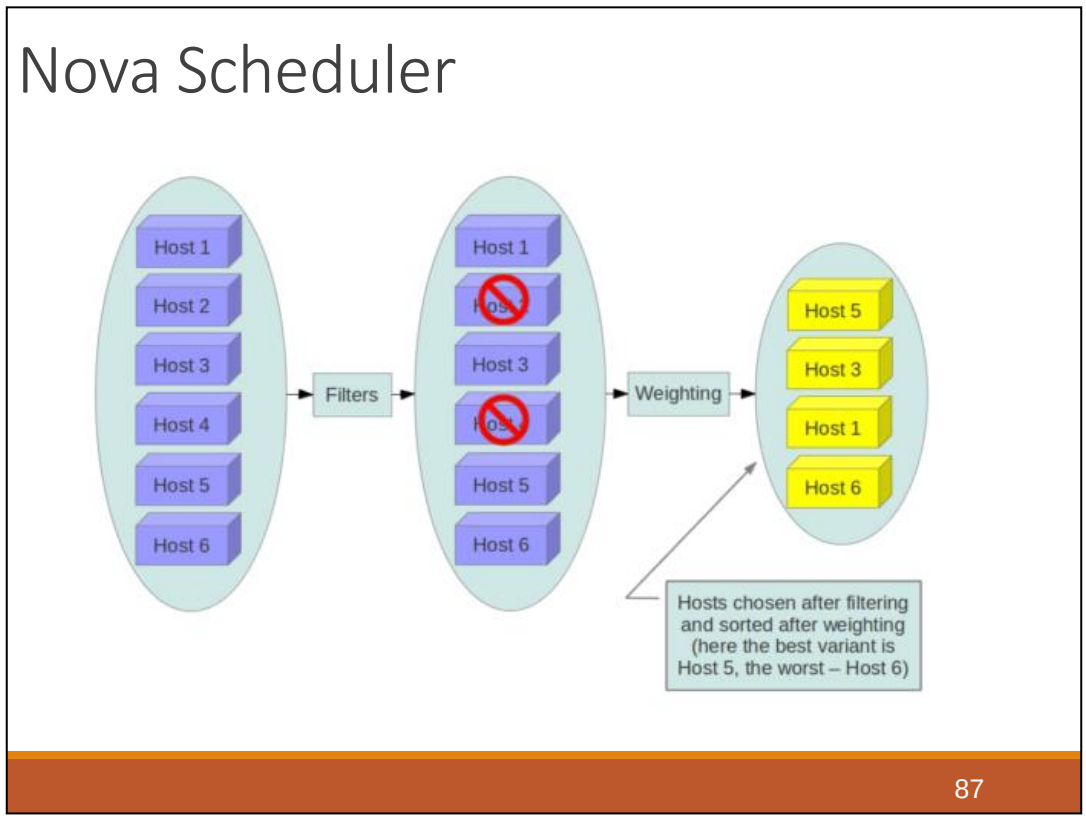
•Remove a shelved instance - Removes the instance from the server; data and resource associations are deleted. If an instance is no longer needed, you can move the instance off the hypervisor in order to minimize resource usage.

$ nova shelve-offload SERVERNAME

"Dettaglio eccessivo"

Task_state

- task_state should represent a transition state, and is precisely associated with one compute API, indicating which task the VM is currently running
- None: no task is currently in progress, BUILDING, IMAGE_SNAPSHOTTING, IMAGE_BACKINGUP, UPDATING_PASSWORD, PAUSING, UNPAUSING, SUSPENDING, RESUMING, DELETING, STOPPING, STARTING, RESCUING, UNRESCUING, REBOOTING, REBUILDING, POWERING_ON, POWERING_OFF, RESIZING, RESIZE_REVERTING, RESIZE_CONFIRMING, SCHEDULING, BLOCK_DEVICE_MAPPING, NETWORKING, SPAWNING, RESIZE_PREP, RESIZE_MIGRATING, RESIZE_MIGRATED, RESIZE_FINISH



https://docs.openstack.org/nova/latest/user/filter-scheduler.html

The **Filter Scheduler** supports *filtering* and *weighting* to make informed decisions on where a new instance should be created. This Scheduler supports working with Compute Nodes only.

During its work Filter Scheduler iterates over all found compute nodes, evaluating each against a set of filters. The list of resulting hosts is ordered by weighers. The Scheduler then chooses hosts for the requested number of instances, choosing the most weighted hosts. For a specific filter to succeed for a specific host, the filter matches the user request against the state of the host plus some extra magic as defined by each filter (described in more detail below).

If the Scheduler cannot find candidates for the next instance, it means that there are no appropriate hosts where that instance can be scheduled.

The Filter Scheduler has to be quite flexible to support the required variety of *filtering* and *weighting* strategies. If this flexibility is insufficient you can implement *your own filtering algorithm*.

There are many standard filter classes which may be used (nova.scheduler.filters):

AllHostsFilter - does no filtering. It passes all the available hosts.

ImagePropertiesFilter - filters hosts based on properties defined on the instance's image. It passes hosts that can support the properties specified on the image used by the instance.

AvailabilityZoneFilter - filters hosts by availability zone. It passes hosts matching the availability zone specified in the instance properties. Use a comma to specify multiple zones. The filter will then ensure it matches any zone specified.

ComputeCapabilitiesFilter - checks that the capabilities provided by the host compute service satisfy any extra specifications associated with the instance type. It passes hosts that can create the specified instance type.

Nova Scheduler

- The **nova-scheduler** handles scheduling for virtual machine instances. The **filter scheduler** is the default scheduler for scheduling virtual machine instances. It supports filtering and weighting to make informed decisions on where a new instance should be created.
  - When it receives a resource request, it first applies filters to determine which host is eligible—the host is either accepted or rejected by a filter.

Nova Scheduler

Example filters include (among others):

- AvailabilityZoneFilter—in the requested availability zone
- ComputeFilter—passes all hosts that are running and enabled
- DiskFilter—is there sufficient disk space?
- CoreFilter—are there sufficient CPU cores available?
- RamFilter—does the host have sufficient RAM available?

Nova Scheduler

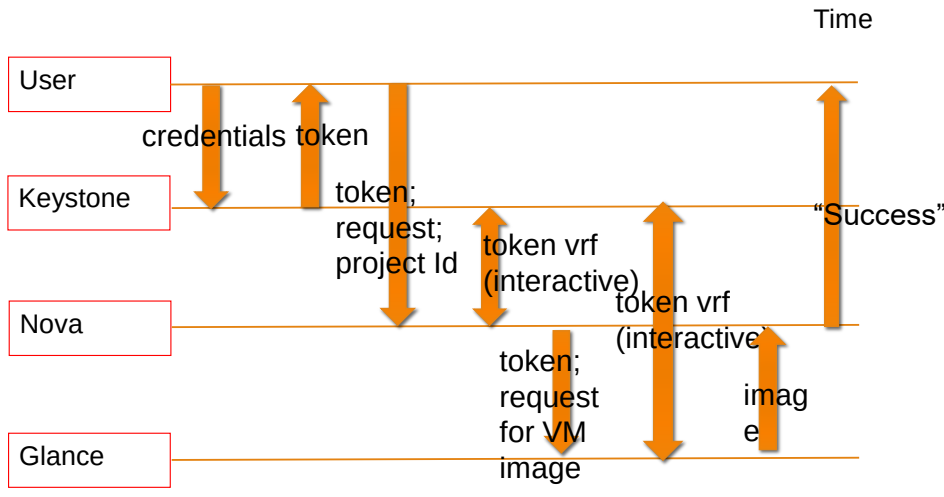
- By default, virtual machine instances are spread evenly across all available hosts, and hosts for new instances are chosen randomly from a set of the N best available hosts.
- Various options can change how this works, for example, one option in the nova.conf (configuration) file allows virtual machine instances to be stacked on one host until that host's resources are used up (set ram_weight_- multiplier in the config file to a negative number). Another option (scheduler_host_subset_size) selects the value N used to select the N best available hosts.

Nova Scheduler

- The filter scheduler weights each host in the list of available hosts. By default, it uses the **RAM Weigher** to determine which host to place the instance on. Using the RAM Weigher, the hosts with large quantities of RAM will be selected first, until you exceed a maximum number of VMs per node.
- If you disable the RAM Weigher, then VMs will be randomly distributed among available hosts (note that each host already was checked by a filter to determine that it had sufficient RAM to run the VM).

Basic---Launching an Instance

Creating/Running a VM without Networks (Neutron), without Persistent Storage (Cinder)

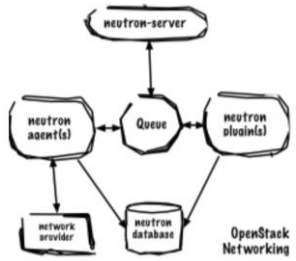


Slide 92

Triggered by a POST request to the */servers/* API endpoint.

OpenStack Neutron

Neutron is an OpenStack project that provides “network connectivity as a service” between interface devices (e.g., vNICs) managed by other OpenStack services (e.g., Nova). It implements the OpenStack’s **Neutron API**



- **Core Use Cases:**

OpenStack Networking mainly interacts with OpenStack Compute to provide networks and connectivity for its instances.

- **Key Capabilities:**

Neutron allows users to configure their own network topology.
It also allows use of advanced network services, including services intended to improve security and quality of service

OpenStack Neutron components communicate with each other using AMQP and a message queue.

https://docs.openstack.org/neutron/zed/

OpenStack Networking (neutron) allows you to create and attach interface devices managed by c

Neutron Features

The following are basic concepts in Neutron:

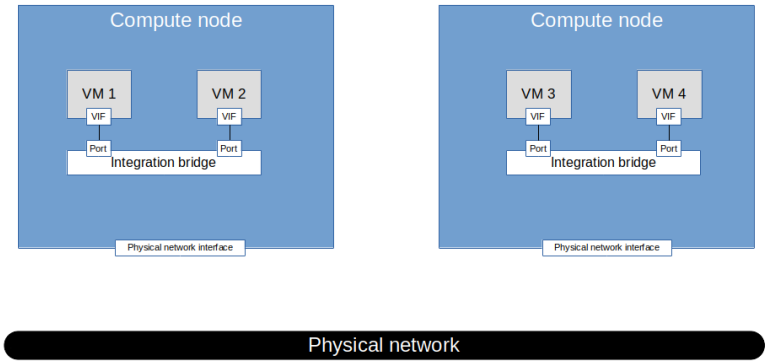
- **Network**—A virtual object that can be created. It provides an independent network for each tenant in a multitenant environment. A network is equivalent to a switch with virtual ports which can be dynamically created and deleted.
- **Port**—A connection port. A router or a VM connects to a network through a port.
- **Subnet**—An address pool that contains a group of IP addresses. Two different subnets communicate with each other through a router.
- **Router**—A virtual router that can be created and deleted. It performs routing selection and data forwarding.

Neutron provides actual layer 2 connectivity to compute instances. Networks in Neutron are layer 2 networks, and if two compute instances are assigned to the same virtual network, they are connected to an actual virtual Ethernet segment and can reach each other on the Ethernet level.

Assume that we are given two virtual machines, call them VM1 and VM2, on the same physical compute node. Our hypervisor will attach a **virtual interface (VIF)** to each of these virtual machines. In a physical network, you would simply connect these two interfaces to ports of a switch to connect the instances. In our case, we can use a **virtual switch / bridge** to achieve this.

OpenStack is able to leverage several bridging technologies. First, OpenStack can of course use the Linux bridge driver to build and configure virtual switches. In addition, Neutron comes with a driver that uses Open vSwitch (OVS).

Neutron Features

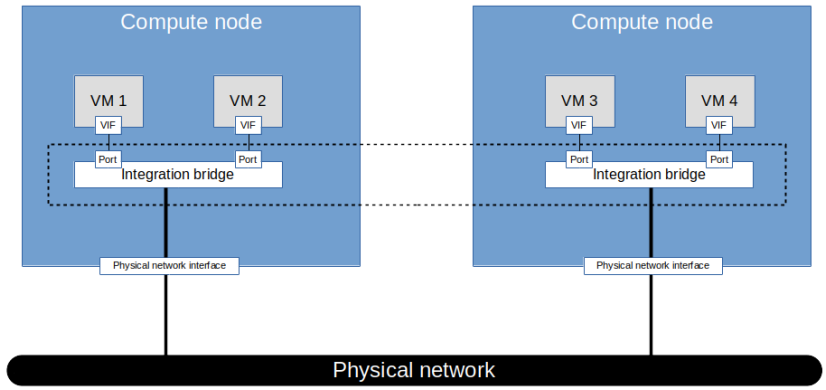


To connect the VMs running on the same host, Neutron could use (and it actually does) an OVS bridge to which the virtual machine networking interfaces are attached. This bridge is called the **integration bridge**. In a typical physical network, this bridge is also connected to a DHCP agent, routers and so forth.

Neutron Features

- But even for the simple case of VMs on the same host, we are not yet done. To operate a cloud at scale, you will need some approach to isolate networks.
- If, for instance, the two VMs belong to different tenants, you do not want them to be on the same network. To do this, Neutron uses **VLANs**. So the ports connecting the integration bridge to the individual VMs are tagged, and there is one VLAN for each Neutron network.

Neutron Features



We need to move on and connect the VMs that are attached to the same network on different hosts. To do this, we will have to use some **virtual networking technology** to connect the integration bridges on the different hosts

Neutron Features

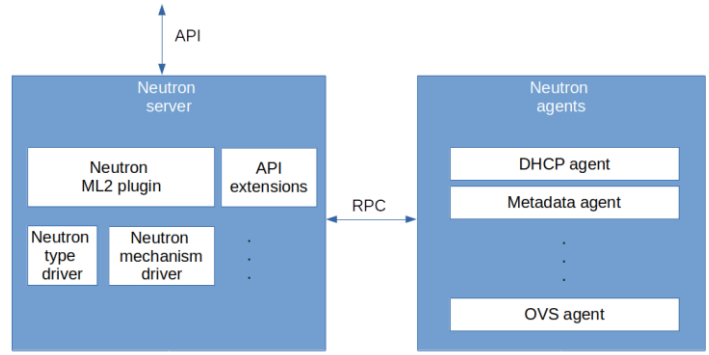
- It is possible simply connect each integration bridge to a physical network device which in turn is connected to the physical network. With this setup, called a **flat network** in Neutron, all virtual machines are effectively connected to the same Ethernet segment. Consequently, there can only be one flat network per deployment.
- The second option we have is to use VLANs to partition the physical network. Neutron would assign a global VLAN ID to each virtual network (which in general is different from the VLAN ID used on the integration bridge) and tag the traffic with the corresponding VLAN ID.
- Finally, we could use tunnels to connect the integration bridges across the hosts. Neutron supports the most commonly used tunneling protocols (**VXLAN**, GRE, Geneve)..

Neutron Architecture

- The Neutron architecture includes the neutron server and the neutron agents
- The main software entity for Neutron is the **neutron-server** daemon.
  - This is basically a python program that starts two critical components:
    - Neutron **REST service**
    - **Neutron Plugins** - **core and service plugins**
  - Accepts and routes **API requests** to the appropriate OpenStack Networking plug-in for action
- The **Neutron RPC service** is also started so that the neutron servers can communicate with the agents. And the RPC service actually loads the Neutron Plugin.

Neutron Architecture

The core functionality of Neutron is provided by a plugin called the **core plugin**.



The Neutron server on the left hand side provides the Neutron API endpoint. Then, there are the components that provide the actual functionality behind the API.

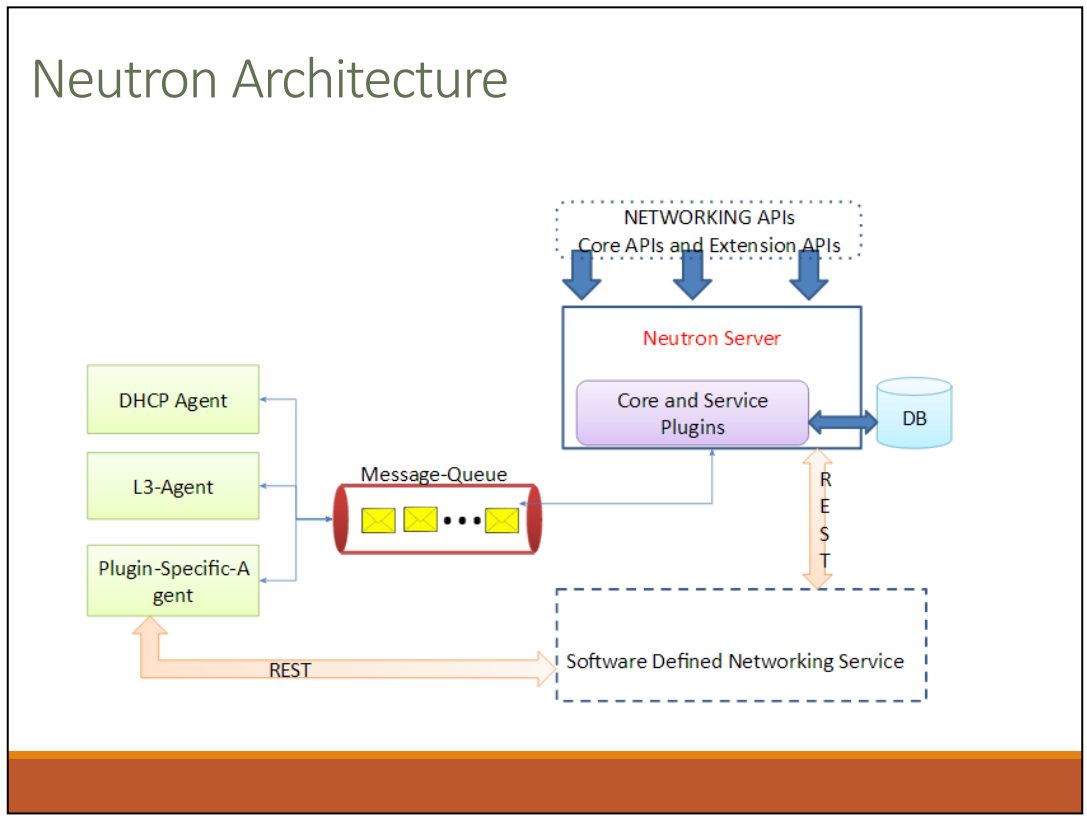
The Neutron API can be extended by API extensions. These extensions (which are again Python classes which are stored in a special directory and loaded upon startup) can be action extensions (which provide additional actions on existing resources), resource extensions (which provide new API resources) or request extensions that add new fields to existing requests.

Neutron comes with agents for additional functionality like DHCP, a metadata server or IP routing. In addition, there are agents running on the compute node to manipulate the network stack there, like the OVS agent or the Linux bridging agent, which correspond to the chosen mechanism drivers.

Neutron Architecture

Neutron has the following components:

- **Neutron plugins** (neutron-*-plugin). maintains configuration data and relationships between routers, networks, subnets, and ports in the Neutron database.
- **Plugins** are pluggable python classes that are invoked while responding to API requests. Neutron plugins are classified into **Core** and **Service plugins**. Core plugin primarily deals with L2 connectivity and IP Address management. On the other hand, Service plugins support services such as Routing (L3), firewall and load-balancing services etc.
- **Note: plugin code is executed as part of Neutron Server on the Controller node.**



OpenStack Networking ships with plugins and agents for Cisco virtual and physical switches, NEC OpenFlow products, Open vSwitch, Linux bridging, and the VMware NSX product.

Plugins are pluggable python classes that are invoked while responding to API requests. Neutron plugins are classified into Core and Service plugins. Core plugin primarily deals with L2 connectivity and IP Address management. On the other hand, Service plugins support services such as Routing (L3), firewall and load-balancing services etc.

Each plug-in that Networking uses has its own concepts. While not vital to operating the VNI and
OpenStack environment, understanding these concepts can help you set up Networking. All Networking
installations use a core plug-in and a security group plug-in (or just the No-Op security group plug-in).
Additionally, Firewall-as-a-Service (FWaaS) is available

Agents

- Provides layer 2/3 connectivity to instances
- Handles physical-virtual network transition
- Handles metadata, etc.

Neutron Features

Neutron Plugins

- Plugins in Neutron allow extension and/or customization of the pre-existing functionality in Neutron. Networking vendors can write plugins that ensures **smooth inter-operability between OpenStack Neutron and vendor-specific software and hardware**. With this approach a rich set of physical and virtual networking resources can be made available to the virtual machines instances.

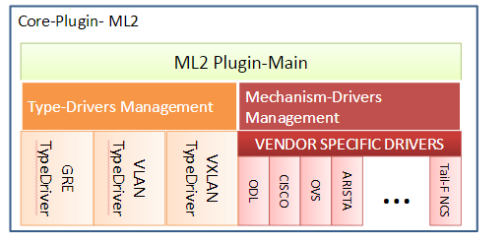
103

The DHCP agent provides DHCP services for virtual networks.
Configuration in `/etc/neutron/dhcp_agent.ini`

The metadata agent provides configuration information such as credentials to instances.
Configuration in `/etc/neutron/metadata_agent.ini`

ML2 Plugin and Drivers

- The ML2 or Modular Layer 2 plugin is bundled with OpenStack. It is an important **core plugin** because it supports wide variety of L2 technologies. More importantly the ML2 plugin allows multiple vendor technologies to co-exist.



- The ML2 plugin supports **Layer 2 technologies** such as VLAN, VXLAN and GRE etc. These technologies are referred to as Type drivers. And these technologies in turn can be implemented using various methods like Open vSwitch or via network hardware. ML2 Plugin allows different implementation methods using mechanism drivers.
- Although ML2 is the sole core plugin, it can support multiple **type drivers and mechanism drivers**. This model provides the flexibility and choice needed for tenants as well as leverage the hardware and software networking resources effectively.

Type Driver: which tells it what type of L2 technology to use when implementing the networking. For example, there is an option of using VLANs, VXLAN, or GRE Tunneling.

Mechanism Driver: which specifies what driver to use to implement the technology.

The ML2 plug-in uses the Linux bridge mechanism to build layer-2 (bridging and switching) virtual networking infrastructure for instances.

Configuration in

`/etc/neutron/plugins/ml2/ml2_conf.ini`

`/etc/neutron/plugins/ml2/linuxbridge_agent.ini`

ML2 Plugin

- The ML2 plugin allows OpenStack Networking to **simultaneously utilize the variety of layer 2 networking technologies** found in complex real-world data centers.
- Each available network type is managed by an ml2 **TypeDriver**.
  - TypeDrivers maintain any needed type-specific network state, and perform provider network validation and tenant network allocation. The ml2 plugin currently includes drivers for the **local, flat, vlan, gre** and **vxlan** network types.
- Each networking mechanism is managed by an ml2 **MechanismDriver**, responsible for taking the information established by the TypeDriver and ensuring that it is properly applied given the specific networking mechanisms enabled.

105

The Modular Layer 2 (ML2) neutron plug-in is a framework allowing OpenStack Networking to simul-
taneously use the variety of layer 2 networking technologies found in complex real-world data centers.
The ML2 framework distinguishes between the two kinds of drivers that can be configured:
• Type drivers
Define how an OpenStack network is technically realized. Example: VXLAN
Each available network type is managed by an ML2 type driver. Type drivers maintain any needed
type-specific network state. They validate the type specific information for provider networks and
are responsible for the allocation of a free segment in project networks.
• Mechanism drivers
Define the mechanism to access an OpenStack network of a certain type. Example: Open vSwitch
mechanism driver.
The mechanism driver is responsible for taking the information established by the type driver and
ensuring that it is properly applied given the specific networking mechanisms that have been en-
abled.
Mechanism drivers can utilize L2 agents (via RPC) and/or interact directly with external devices

or controllers.
Multiple mechanism and type drivers can be used simultaneously to access different ports of the same
virtual network

To enable mechanism drivers in the ML2 plug-in, edit the /etc/neutron/plugins/ml2/ml2_conf.ini file on the neutron server.

ML2 Plugin

type driver / mech driver	Flat	VLAN	VXLAN	GRE
Open vSwitch	yes	yes	yes	yes
Linux bridge	yes	yes	yes	no
SRIOV	yes	yes	no	no
MacVTap	yes	yes	no	no
L2 population	no	no	yes	yes

vSphere supports **Single Root I/O Virtualization (SR-IOV).** You can use SR-IOV for networking of virtual machines that are latency sensitive or require more CPU resources.

SR-IOV is a specification that allows a single Peripheral Component Interconnect Express (PCIe) physical device under a single root port to appear as multiple separate physical devices to the hypervisor or the guest operating system.

Macvtap is a new device driver meant to simplify virtualized bridged networking. It replaces the combination of the tun/tap and bridge drivers with a single module based on the macvlan device driver. A macvtap endpoint is a character device that largely follows the tun/tap ioctl interface and can be used directly by kvm/qemu and other hypervisors that support the tun/tap interface. The endpoint extends an existing network interface, the lower device, and has its own mac address on the same ethernet segment. Typically, this is used to make both the guest and the host show up directly on the switch that the host is connected to.

L2 population is a special mechanism driver that optimizes BUM (Broadcast, unknown destination address, multicast) traffic in the overlay networks VXLAN and GRE. It needs to be used in conjunction with either the Linux bridge or the Open vSwitch mechanism driver and cannot be used as standalone mechanism driver. For more information, see the *Mechanism drivers* section below.

Specialized

- Open source
- External open source mechanism drivers exist as well as the neutron integrated reference implementations. Configuration of those drivers is not part of this document. For example:
  - OpenDaylight
  - OpenContrail
- Proprietary (vendor)
- External mechanism drivers from various vendors exist as well as the neutron integrated reference implementations.

no

Vendor Specific Mechanism Drivers

The set of drivers included in the main Neutron distribution and supported by the Neutron community include:

Open vSwitch

Cisco UCS/Nexus

Cisco Nexus1000v

Linux Bridge

Modular Layer 2

Nicira Network Virtualization Platform (NVP)

Ryu OpenFlow Controller

NEC OpenFlow

Big Switch Controller Plugin

Cloudbase Hyper-V

MidoNet

Brocade Neutron Plugin

PLUMgrid

Mellanox Neutron Plugin

Embrane Neutron Plugin

- IBM SDN-VE
- CPLANE NETWORKS
- Nuage Networks
- OpenContrail
- Lenovo Networking
- Avaya Neutron Plugin

Additional plugins are available from other sources:

- Extreme Networks Plugin
- Ruijie Networks Plugin
- Juniper Networks Neutron Plugin
- Calico Neutron Plugin (docs)
- BNC Plugin (docs)

no

Service Plugins

- Historically, Neutron supported the following advanced services:
  - **FWaaS** (*Firewall-as-a-Service*): runs as part of the L3 agent.
  - **VPNaaS** (*VPN-as-a-Service*): derives from L3 agent to add VPNaaS functionality.
- **Service plugins** implement different additional services, some of them already mentioned, including load balancing (Load Balancing as a Service or **LBaaS**), VPNs (**VPNaaS**), firewalls (**FWaaS**), metering, Trunk.

no, se vuoi leggi ma nulla di nuovo

Neutron Agents

- While the Neutron server acts as the centralized controller, the actual networking related commands and configuration are executed on the Compute nodes and **agents** are the entities that implement the actual networking changes on these nodes. Agents receive messages and instructions from the Neutron server (via plugins or directly) on the message bus.
- Since Agents are responsible for the implementation of networking, they are **closely associated with specific technologies** and the corresponding plugins.
- **Plugin agent** (neutron-*-agent)—Processes data packets on virtual networks. The choice of plug-in agents depends on Neutron plug-ins. A plug-in agent interacts with the Neutron server and the configured Neutron plug-in through a message queue.
- **DHCP agent** (neutron-dhcp-agent)—Provides DHCP services for tenant networks.
- **L3 agent** (neutron-l3-agent)—Provides Layer 3 forwarding services to enable inter-tenant communication and external network access by "routers" that connect to L2 networks.
  - It is responsible for providing layer 3 and NAT forwarding to gain external access for virtual machines on tenant networks.

Nutron API Concepts

The Neutron v2 API manages three kind of entities:

- **Network**, representing isolated virtual Layer-2 domains; a network can also be regarded as a virtual (or logical) switch.
- **Subnet**, representing IPv4 or IPv6 address blocks from which IPs to be assigned to VMs on a given network are selected.
- **Port**, representing virtual (or logical) switch ports on a given network.
- **Router**, representing a virtual device taking care of L3 operations including floating IP.

All entities support the basic CRUD operations with POST/GET/PUT/DELETE verbs, and have an auto-generated unique identifier.

CRUD: Create Read Update Delete

Network Configuration

- To configure rich network topologies, you can create and configure **networks and subnets** and instruct other OpenStack services like Compute to attach virtual devices to ports on these networks.
- OpenStack Compute (**Nova**) is a consumer of OpenStack Networking to provide connectivity for its instances.
- There are two types of network, **project (or self-service)** and **provider** networks. It is possible to share any of these types of networks among projects as part of the network creation process.

Provider networks

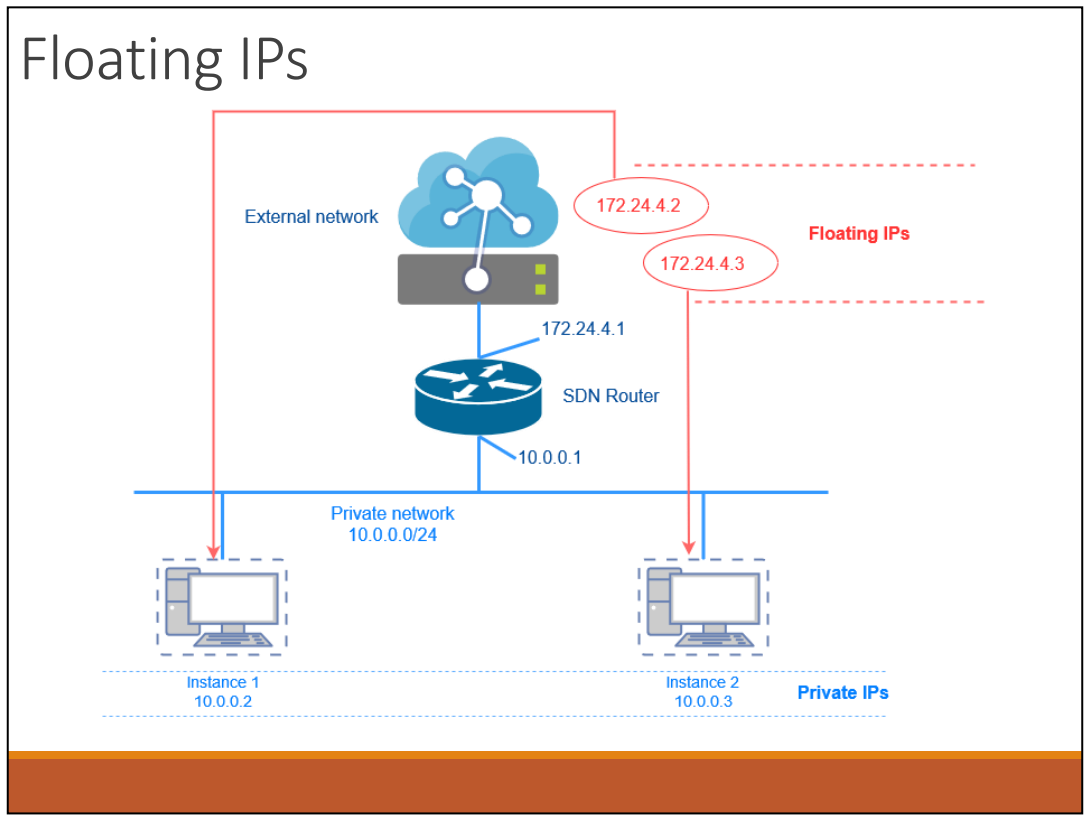
- **Provider networks** offer layer-2 connectivity to instances with optional support for DHCP and metadata services. These networks connect, or map, to existing layer-2 networks in the data center, typically using VLAN (802.1q) tagging to identify and separate them.
- Provider networks generally offer simplicity, performance, and reliability at the cost of flexibility. By default only administrators can create or update provider networks because they require configuration of physical network infrastructure.

Self Service networks

- Self-service networks primarily enable general (non-privileged) projects to manage networks without involving administrators. These networks are entirely virtual and require virtual routers to interact with provider and external networks such as the Internet. Self-service networks also usually provide DHCP and metadata services to instances.
- In most cases, self-service networks use overlay protocols such as VXLAN or GRE because they can support many more networks than layer-2 segmentation using VLAN tagging (802.1q).

Self Service networks

- IPv4 self-service networks typically use **private IP address** ranges and interact with provider networks via source NAT on virtual routers. **Floating IP** addresses enable access to instances from provider networks via destination NAT on virtual routers. **IPv6** self-service networks always use public IP address ranges and interact with provider networks via virtual routers with static routes.
- The Networking service implements routers using a layer-3 agent that typically resides at least one network node. Contrary to provider networks that connect instances to the physical network infrastructure at layer-2, self-service networks must traverse a layer-3 agent.



Instance 1 and instance 2 and they have IP addresses that come from the private network (behind a NAT router). Everything that is behind the NAT router cannot be addressed directly so the 10.0.0.2 and 10.0.0.3 IP addresses cannot be accessed directly from the Internet.

That is why in OpenStack Software Defined Networking, to make instances accessible, these instances need to be allocated floating IP addresses.

Floating IPs are IP addresses are exposed at the external side of the NAT router which is the SDN router. So, for external traffic to reach instance 1 and instance 2, the external traffic would go directly to the floating IP addresses of the instances. In the above example, traffic arrives 172.24.4.2 and 172.24.4.3 will be forwarded to 10.0.0.2 and 10.0.0.3 respectively.

Used for communication with networks outside the cloud, including the Internet

A floating IP address and a private IP address can be used at the same time on a NIC (Network Interface Card)

NOT automatically allocated to instances by default, they need to be attached to instances manually

nah, già detto prima

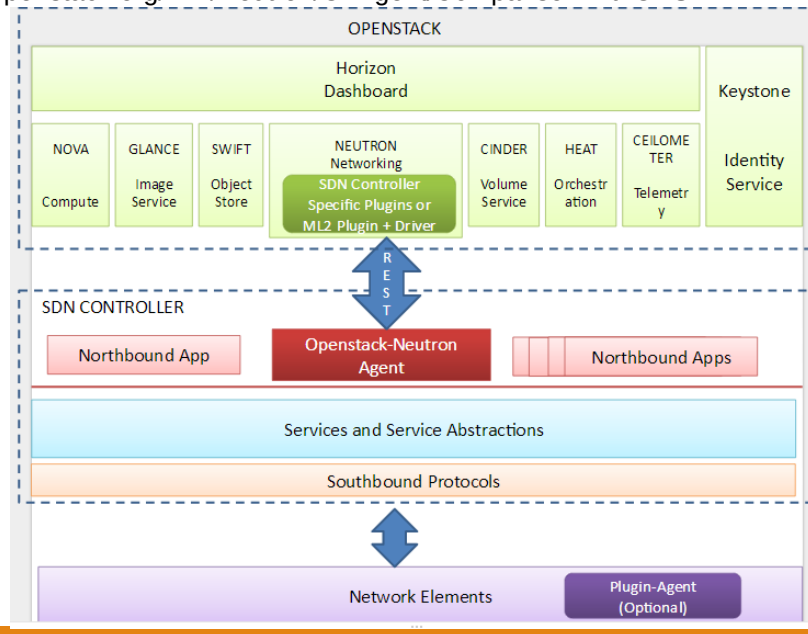
OpenStack SDN

- Softwared-defined networking was introduced to both overcome the deficiencies of Neutron and to provide support for multiple network virtualization technologies (a centralized control plane creating isolated tenant virtual networks) and approaches. With the integration on SDN, Neutron is expected to support the dynamic nature of large-scale, high-density, multi-tenant cloud environments.
- OpenStack Neutron, with its plugin architecture, provides the ability to integrate SDN controllers into the OpenStack. This integration of SDN controllers into Neutron using plugins provides centralized management, and also facilitates the network programmability of OpenStack networking using APIs.
- SDN controllers like OpenDaylight, Ryu, and Floodlight use either specific plugins or the ML2 plugin with the corresponding mechanism drivers, to allow communication between Neutron and the SDN controller.

nah --> Ecco l'architettura di SDN, ma spiegato già prima

OpenStack SDN

https://wiki.openstack.org/wiki/Neutron/OFAgent/ComparisonWithOVS

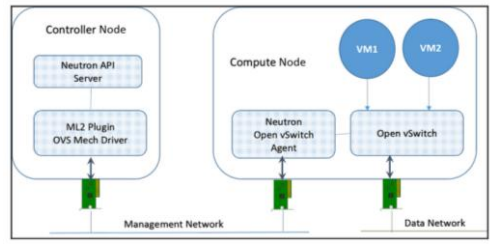


With this integration of OpenStack Neutron and SDN controllers, the changes to the network and network elements are also triggered from the OpenStack user, which are translated into Neutron APIs, and handled by neutron plugins and corresponding agents running in SDN controllers. For example, OpenDaylight interacts with Neutron by using the ML2 plugin present on the network node of Neutron via the REST API using northbound communication.

"Livello di dettaglio eccessivo, ma vabbè ormai ve lo dico"
[min 49 se ti interessa, del 09/04]

OpenStack SDN

Open Vswitch Plugin and Driver



- The **Neutron API server** receives commands via **REST API clients**, such as Horizon.
- The **API server** invokes the **ML2 Plugin** to process the request.
- The **ML2 Plugin** passes the request to the configured **OVS mechanism driver**.
- The **OVS mechanism driver** contructs an RPC message and directs it **towards the OVS agent** on the Compute Node over the Management Network Interface.
- The **OVS agent** in the Compute Node communicates witht the **local OVS** instance to program it according to the instructions on the received command.
- Currently the OVS agent integrates the **RYU controller**.

From Newton on, by default Neutron uses the native interfaceof OVSDB and OpenFlow. The use of this interface allows Neutron to callryu.base.appmanager during operation, and by default the native interfacewill have the Ryu controller listen on 127.0.0.1:6633 which can be howevermodified to point to another address.The integration among Ryu and OpenStack is very basic: however, it ispossible to load applications through the appmanager that is loaded by Neutron: therefore, the possible scenarios that can take place are infinite, becauseit simply needs the administrator to run the SDN application that he or shewants in order to manage the cloud platform in addition to or in a differentway with respect to the default Neutron mechanisms.

Prior to OpenStack Newton, the neutron-openvswitch-agent used “ovs-ofctl”ofinterface driver by default to communicate with the Open vSwitch. FromNewton on instead, the default implementation for ofinterface has become thenovel “native”, that mostly eliminates spawning ovs-ofctl to slightly improvethe networking performance. This is an alternative OpenFlow implementation,implemented usingRyu SDN controllerofproto python library from Ryu SDNFramework. This solution indeed uses Ryu to inject OpenFlow rules whenrequested (e.g. a new virtual instance is being created or an old one is deleted)instead of building a “ovs-ofctl add/del-flow”.The consequences are:•The implemented OpenFlow rules are switched to OpenFlow 1.3, ratherthan OpenFlow 1.0;•The OvS-agent acts as an OpenFlow controller perfectly integrated with the OpenStack platform;•The OvS of the node is configured to connect to the controller.

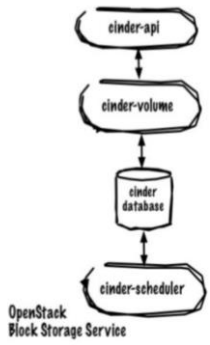
90CHAPTER 4. NEUTRON NETWORKINGAmong the benefits, it is also possible to state:•Reduction of the overhead related to the invocation of the ovs-ofctl command (and its associated rootwrap);•Easier dealing with a future use of OpenFlow asynchronous messages(e.g. Packet-In, Port-Status, etc.)⇒Introduction of a SDN controllerinside the OpenStack platform, integration among the platform and theframework⇒Possibility to directly manage the OpenStack networkingby managing Ryu;•Simpler XenAPI integration.

OpenStack Data Storage

- Ephemeral
  - Each running VM receives some ephemeral storage, 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 VM.
- Block Storage
  - It is performed by **Cinder**, which attaches storage volumes to a VM. A storage volume hasits own file system. The server running Cinder can have storage volumes located on its physical disks, or can be located on other physical disks.
- Object Storage
  - It is performed using **Swift**. It stores data as binary objects that are retrieved and written using HTTP commands (GET, PUT, etc.). Swift stores objects on object servers. Swift is best used for storing unstructured data, such as email, images, audio, and video, etc.

OpenStack Volume Storage: Cinder

- Cinder provides block storage service



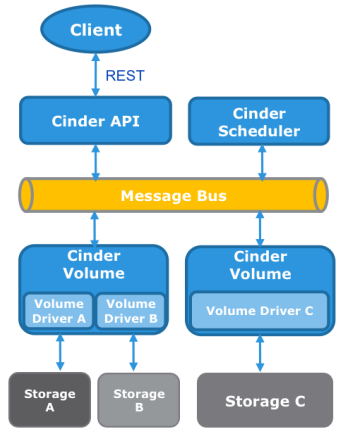
- **Core Use Cases:**

To implement services and libraries to provide on demand, self-service access to Block Storage resources.

- **Key Capabilities:**

It is designed to present storage resources to end users that can be consumed by the OpenStack Compute Project (Nova). The short description of Cinder is that it virtualizes the management of block storage devices and provides end users with a self service API to request and consume those resources without requiring any knowledge of where their storage is actually deployed or on what type of device.

OpenStack Cinder



- All features of Cinder are exposed via a REST
- API that can be used to build more complicated logic or automation with Cinder. This can be consumed directly or via **various SDKs**.

- The **Block Storage API and scheduler services typically run on the controller nodes**. Depending upon the drivers used, the volume service can run on controller nodes, compute nodes, or standalone storage nodes.

Cinder Components

The Block Storage service consists of the following components:

- **cinder-api**
  - Accepts API requests, and routes them to the cinder-volume for action.
- **cinder-volume**
  - **manages Block Storage devices, specifically the back-end devices themselves**. The cinder-volume service responds to read and write requests sent to the Block Storage service to maintain state. It can interact with a variety of storage providers through a driver architecture.
  - Interacts directly with the other services and processes, such as the cinder-scheduler, through a message queue
- **cinder-scheduler daemon**
  - Selects the optimal storage provider node on which to create the volume. A similar component to the nova-scheduler. Depending upon your configuration, this may be simple round-robin scheduling to the running volume services, or it can be more sophisticated through the use of the Filter Scheduler.

Back-end Storage Devices - the Block Storage service requires some form of back-end storage that the service is built on. The default implementation is to use LVM on a local volume group named "cinder-volumes." In addition to the base driver implementation, the Block Storage service also provides the means to add support for other storage devices to be utilized such as external Raid Arrays or other storage appliances. These back-end storage devices may have custom block sizes when using KVM or QEMU as the hypervisor.

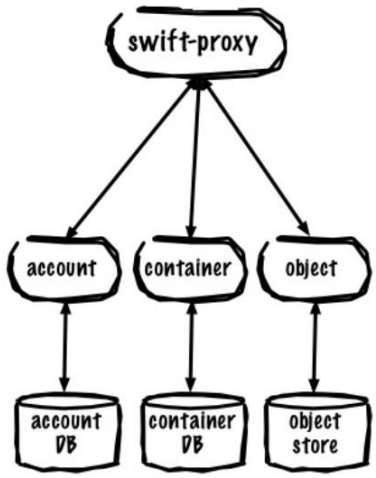
Users and Tenants (Projects) - the Block Storage service can be used by many different cloud computing consumers or customers (tenants on a shared system), using role-based access assignments. Roles control the actions that a user is allowed to perform. In the default configuration, most actions do not require a particular role, but this can be configured by the system administrator in the cinder policy file that maintains the rules.

The Filter Scheduler is the default and enables filters on things like Capacity, Availability Zone, Volume Types, and Capabilities as well as custom filters.

Cinder Components

- **cinder-backup daemon**
  - The cinder-backup service provides backing up volumes of any type to a backup storage provider. Like the cinder-volume service, it can interact with a variety of storage providers through a driver architecture. The installation is optional.
- **Messaging queue**
  - Routes information between the Block Storage processes.

OpenStack Block Storage: Swift



This service provides a simple storage service for applications using RESTful interfaces, providing maximum data availability and storage capacity.

Swift is composed of three major parts:

- **swift-proxy,**
- **storage servers**
  - **account**
  - **container**
  - **object**
- **consistency servers**

Swift Architecture: Proxy Server

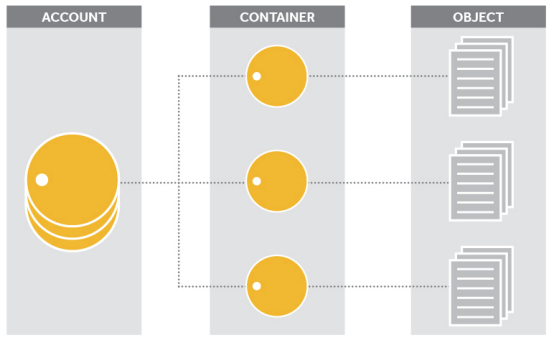
- The **proxy server** is an HTTP server that implements swift's RESTful API.
- As the only system in the swift cluster that communicates with clients, the proxy is responsible for coordinating with the storage servers and replying to the client with appropriate messages.

un po' ridondante

Swift Architecture: Storage Servers

In Swift 3 categories of things to store exist: ***accounts***, ***containers*** and ***objects***.

An **account** is a user account. An account contains **containers** (the equivalent of Amazon S3's buckets). Each container can contains user-defined key and values (just like a hash table or a dictionary): values are what Swift call **objects**.



A **storage URL** in Swift for an object looks like this:

https://swift.example.com/v1/account/container/object

A foundational premise of Swift is that requests are made via HTTP using a RESTful API. All requests sent to Swift are made up of at least three parts:

- **HTTP verb (e.g., GET, PUT, DELETE)**
- **Authentication information**
- **Storage URL**
- Optional: any data or metadata to be written

The HTTP verb provides the action of the request. I want to PUT this object into the cluster. I want to GET this account information out of the cluster, etc.

The authentication information allows the request to be fulfilled.

A storage URL in Swift for an object looks like this: https://swift.example.com/v1/account/container/object

The storage URL has two basic parts: cluster location and storage location. This is because the storage URL has two purposes. It's the cluster address where the request should be sent and it's the location in the cluster where the requested action should take place.

Using the example above, we can break the storage URL into its two main parts:

- Cluster location: swift.example.com/v1/
- Storage location (for an object): /account/container/object

A storage location is given in one of three formats:

- /account

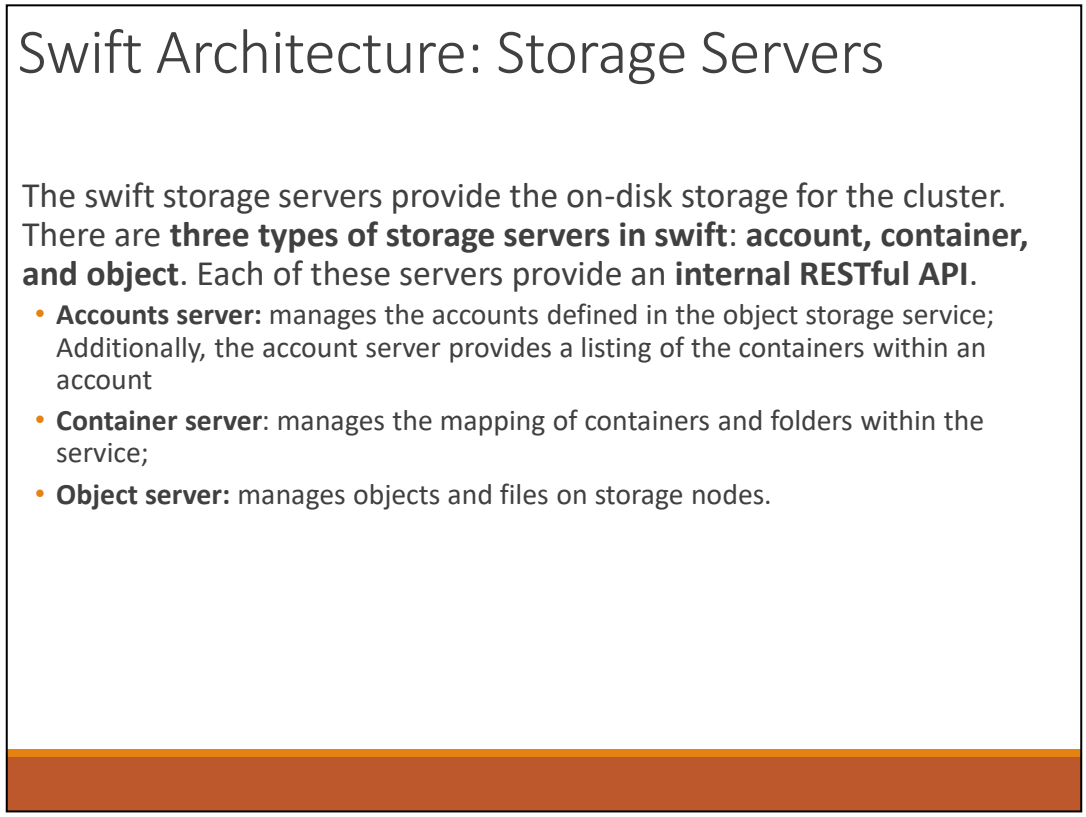
- The account storage location is a uniquely named storage area that contains the metadata (descriptive information) about the account itself as well as the list of containers in the account.
- Note that in Swift, an account is not a user identity. When you hear account, think storage area.

•/account/container

- The container storage location is the user-defined storage area within an account where metadata about the container itself and the list of objects in the container will be stored.

•/account/container/object

"Slide molto ridondate"
[questa neanche la guarda]



Account Layer

The account server process handles requests regarding metadata for the individual accounts or the list of the containers within each account. This information is stored by the account server process in SQLite databases on disk.

Container Layer

The container server process handles requests regarding container metadata or the list of objects within each container. It's important to note that the list of objects doesn't contain information about the location of the object, simply that it belong to a specific container. Like accounts, the container information is stored as SQLite databases.

Object Layer

The object server process is responsible for the actual storage of objects on the drives of its node. Objects are stored as binary files on the drive using a path that is made up in part of its associated partition and the operation's timestamp. The timestamp is important as it allows the object server to store multiple versions of an object. The object's metadata (standard and custom) is stored in the file's extended attributes (xattrs) which means the data and metadata are stored together and copied as a single unit.

Swift Architecture: Storage Servers

- The account and container servers provide **namespace partitioning and listing functionality**. They are implemented as databases (SQLite) on disk, and as other entities in Swift, they are replicated to multiple availability zones within the swift cluster.
- Swift is designed for **multi-tenancy**. Users are generally given access to a single swift account within a cluster, and they have complete control over that unique namespace.

Consistency Management

In theoretical computer science, the **CAP theorem** states that it is impossible for **a distributed data store** to **simultaneously** provide more than two out of the following three guarantees:

Consistency: Every read receives the most recent write or an error

Availability: Every request receives a (non-error) response, regardless it contains the most recent write

Partition tolerance: The system continues to operate despite any number of communication breakdowns between nodes in the system

Consistency Services

A key aspect of Swift is that it acknowledges that failures happen and is built to work around them. When account, container or object server processes are running on node, it means that data is being stored there. That means consistency services will also be running on those nodes to ensure the integrity and availability of the data.

Consistency Management

- Swift aims to **availability** and **partition tolerance** and dropped **consistency**. Hence it is always possible to get data, they will be dispersed on many places, but it is possible to receive an old version of them (or no data at all) in some odd cases (as server overload or failure).
- This compromise is made to allow maximum availability and scalability of the storage platform.
- For this reason, some mechanisms are built into Swift to minimize the potential data inconsistency window: they are responsible for data replication and consistency.

Consistency Management

- Storing data on disk and providing a RESTful API is quite easy. The hard part is handling failures. Swift's **consistency servers** are responsible for finding and correcting errors caused by both data corruption and hardware failures.
- Swift introduces three background processes to solve the problem of data consistency: **Auditor, Updater,** and **Replicator**.

The two main consistency services are auditors and replicators. There are also a number of specialized services that run in support of individual server process, e.g., the account reaper that runs where account server processes are running.

Consistency Management

- **Auditors** continually *scan the disks* to ensure that the data stored on disk has not suffered any bit or file system corruption. If an error is found, the corrupted object is moved to a quarantine area, and replication is responsible for replacing the data with a known good copy.
- **Updaters** ensure that *account and container listings are correct*. The *object updater* is responsible for keeping the object listings in the containers correct, and the *container updaters* are responsible for keeping the account listings up-to-date
- **Replicators** ensure that the *data stored in the cluster is where they should be* and that enough copies of the data exist in the system. Generally, replicators are responsible for repairing any corruption or degraded durability in the cluster.

Auditor is responsible for data auditing. It checks the integrity of Account, Container and Object by continuously scanning the disk. If data is found to be damaged, Auditor will isolate the file and obtain a complete copy from other nodes to replace it. , And the task of this replica is completed by the Replicator. In addition, in the rebalance operation of Ring, the Replicator is required to complete the actual data migration work. When the Object is deleted, the Replicator also completes the actual deletion operation.

Updater is responsible for handling those Account or Container update operations that fail due to insufficient load and other reasons. The Updater scans the Container or Object data on the local node, and then checks whether there are records of these data on the corresponding Account or Container node, and if not, pushes the records of these data to the Account or Container node. Only Container and Object have corresponding Updater processes, and there is no Account Updater process.

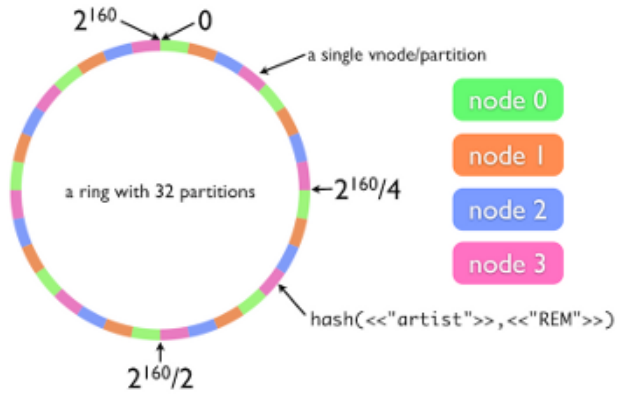
The implementation process of these three processes is similar. Here we take the Replicator process of Account as an example.

There are two types of Replicators in Swift: One is Database Replicator, for the two types of data in the form of a database, Account and Container, and the other is Object Replicator, which serves Object data.

Consistent hashing

Swift uses the principle of consistent hashing. It builds what it calls a *ring*. A ring represents the space of all possible computed hash values divided in equivalent parts. Each part of this space is called a *partition*.

Example:



In order to store some objects and distribute them on 4 nodes, you would split your hash space in 4. You would have 4 partitions, and computing *hash(object) modulo 4* would tell you where to store your object: on node 0, 1, 2 or 3.

But since you want to be able to extend your storage cluster to more nodes without breaking the whole hash mapping and moving everything around, you need to build a lot more partitions. Let's say we're going to build 2^{10} partitions. Since we have 4 nodes, each node will have $2^{10} \div 4 = 256$ partitions. If we ever want to add a 5^{th} node, it's easy: we just have to re-balance the partitions and move 1⁄5 of the partitions from each node to this 5^{th} node. That means all our nodes will end up with $2^{10} \div 5 \approx 204$ partitions. We can also define a *weight* for each node, in order for some nodes to get more partitions than others.

With 2^{10} partitions, we can have up to 2^{10} nodes in our cluster.

Data duplication

To assure availability and partitioning, by default **Swift stores 3 copies of every objects**, but that's configurable.

In that case, we need to store each partition defined above not only on 1 node, but on 2 others. So Swift adds another concept: **zones**. A zone is an isolated space that does not depends on other zone, so in case of an outage on a zone, the other zones are still available. Concretely, a zone is likely to be a disk, a server, or a whole cabinet, depending on the size of your cluster. It's up to you to choose anyway.

Consequently, each partitions has not to be mapped to 1 host only anymore, but to N hosts. Each node will therefore store this number of partitions:

number of partition stored on one node = number of replicas × total number of partitions ÷ number of node

Examples:

We split the ring in 2^{10} = 1024 partitions. We have 3 nodes. We want 3 replicas of data.
→ Each node will store a copy of the full partition space:
$3 \times 2^{10} \div 3 = 2^{10}$ = 1024 partitions. We split the ring in 2^{11} = 2048 partitions. We have 5 nodes. We want 3 replicas of data.
→ Each node will store $2^{11} \times 3 \div 5 \approx 1129$ partitions. We split the ring in 2^{11} = 2048 partitions. We have 6 nodes. We want 3 replicas of data.
→ Each node will store $2^{11} \times 3 \div 6$ = 1024 partitions.

Multiple Ring Operation

- Since in Swift 3 categories of things to store exist, ***accounts***, ***containers*** and ***objects***, Swift makes use of 3 different and independent rings to store its 3 kind of things (*accounts*, *containers* and *objects*).
- Internally, the two first categories are stored as SQLite databases, whereas the last one is stored using regular files.
- Note that these 3 rings can be stored and managed on 3 completely different set of servers.

OpenStack RESTful Interface

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

- The OpenStack Dashboard.
- Using REST clients to send appropriate HTTP commands
- Using Python with the Python libraries provided with OpenStack
- Using the command-line clients
- Using HOT scripts

Example: Accessing Keystone for Authentication

CURL Command

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

my_credentials file:

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

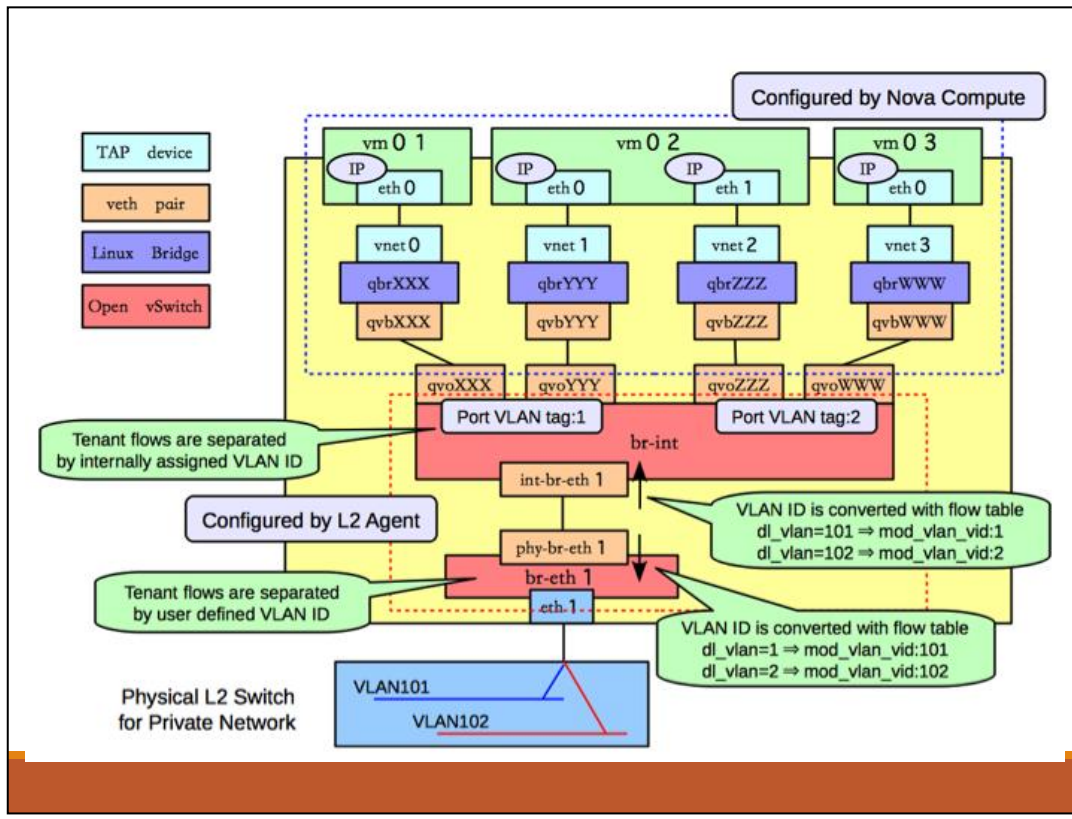
137

Other examples - https://docs.openstack.org/keystone/rocky/api_curl_examples.html

Example: Authentication by Python

- Use of the v3 interface. When you're using the v3 interface, you should use sessions.
- When using sessions, a session object stores your credentials. The session object is passed to a client so the client can do appropriate OpenStack-related actions.
- After a session is established, the session object is responsible for handling authentication. When a request that needs authentication is sent to the session, then the session is responsible for requesting a token from the authentication plugin (or using an existing token). An OpenStack authentication plugin is an implementation of a method of authentication. Identity plugins are those associated with Keystone.

```
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
```



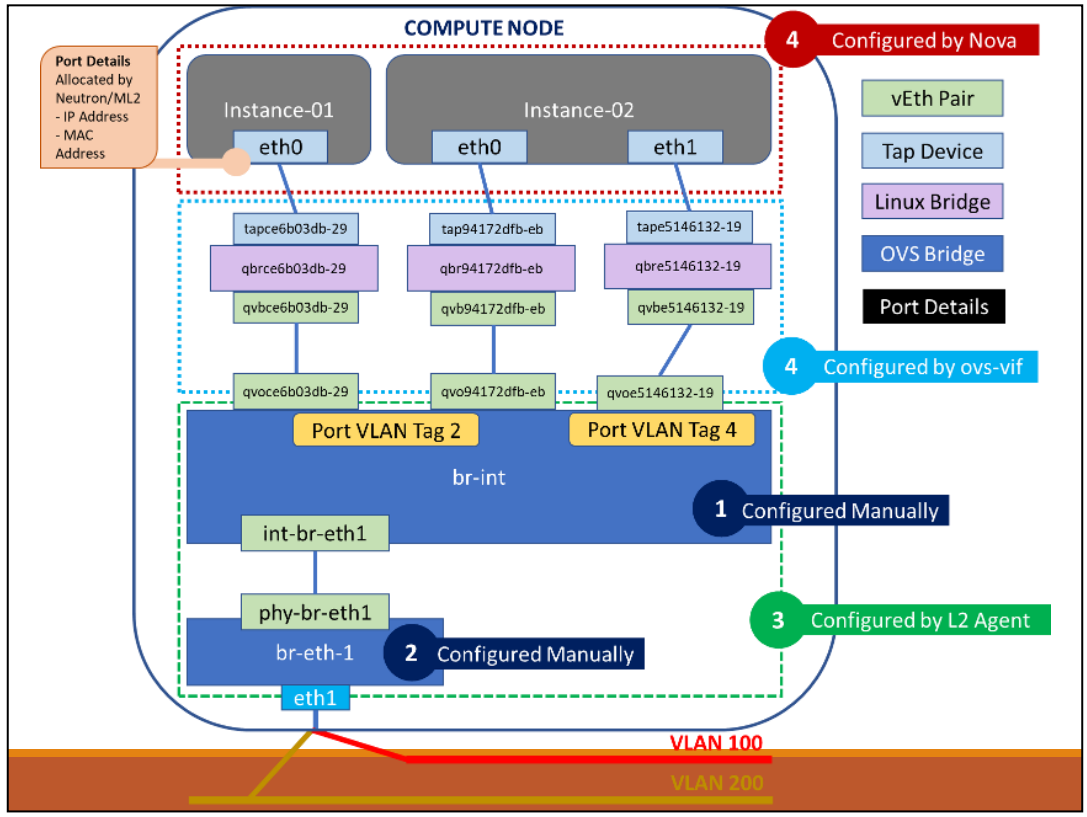
https://docs.openstack.org/neutron/latest/contributor/internals/

https://docs.openstack.org/neutron/latest/contributor/internals/openvswitch_agent.html

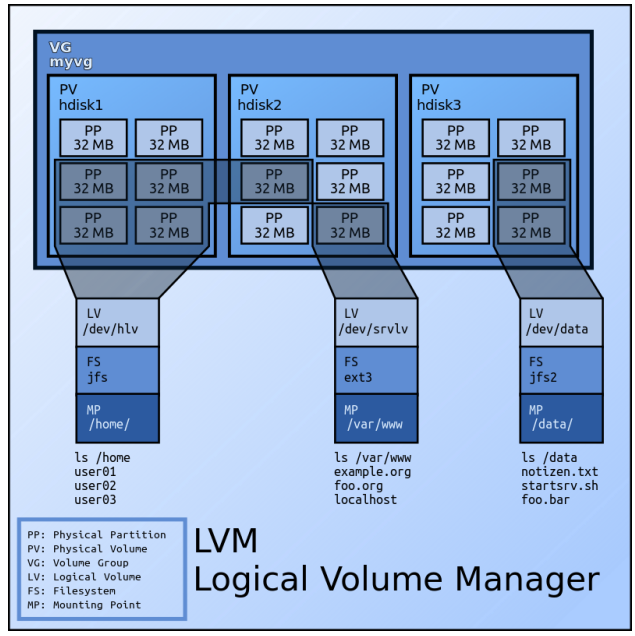
TUN/TAP provides packet reception and transmission for user space programs. It can be seen as a simple Point-to-Point or Ethernet device, which, instead of receiving packets from physical media, receives them from user space program and instead of sending packets via physical media writes them to the user space program.

In order to use the driver a program has to open /dev/net/tun and issue a corresponding ioctl() to register a network device with the kernel. A network device will appear as tunXX or tapXX, depending on the options chosen. When the program closes the file descriptor, the network device and all corresponding routes will disappear.

Depending on the type of device chosen the userspace program has to read/write IP packets (with tun) or ethernet frames (with tap). Which one is being used depends on the flags given with the ioctl().



https://www.techblog.moebius.space/posts/2018-02-17-openstack-neutron-understanding-l2-networking-and-port-binding/


VG
myvg
PV
hdisk1
PV
hdisk2
PV
hdisk3
PP
32 MB
LV
/dev/hlv
FS
jfs
MP
/home/
LV
/dev/srvlv
FS
ext3
MP
/var/www
LV
/dev/data
FS
jfs2
MP
/data/
ls /home
user01
user02
user03
ls /var/www
example.org
foo.org
localhost
ls /data
notizen.txt
startsrv.sh
foo.bar
PP: Physical Partition
PV: Physical Volume
VG: Volume Group
LV: Logical Volume
FS: Filesystem
MP: Mounting Point
LVM
Logical Volume Manager