Inside OpenStack

You have a cloud: now what?

The Plan for today…

Part I: Franz (~80 min or so)

- First steps after the cloud is up and running
- Creation of: images, flavors, networks, VMs, security groups, volumes…
- Hands-on session: play with our dev-cloud

Part II: Leonard (~30 min or so)

- Internal organisation of groups, roles, projects
- Synchronisation with SUPR
- Accounting

The cloud is up – first steps

- Kolla-ansible provides script init-runonce
  - Great starting point to learn how to use the openstack CLI
  - Sets up the first project + image + network + flavor + quota

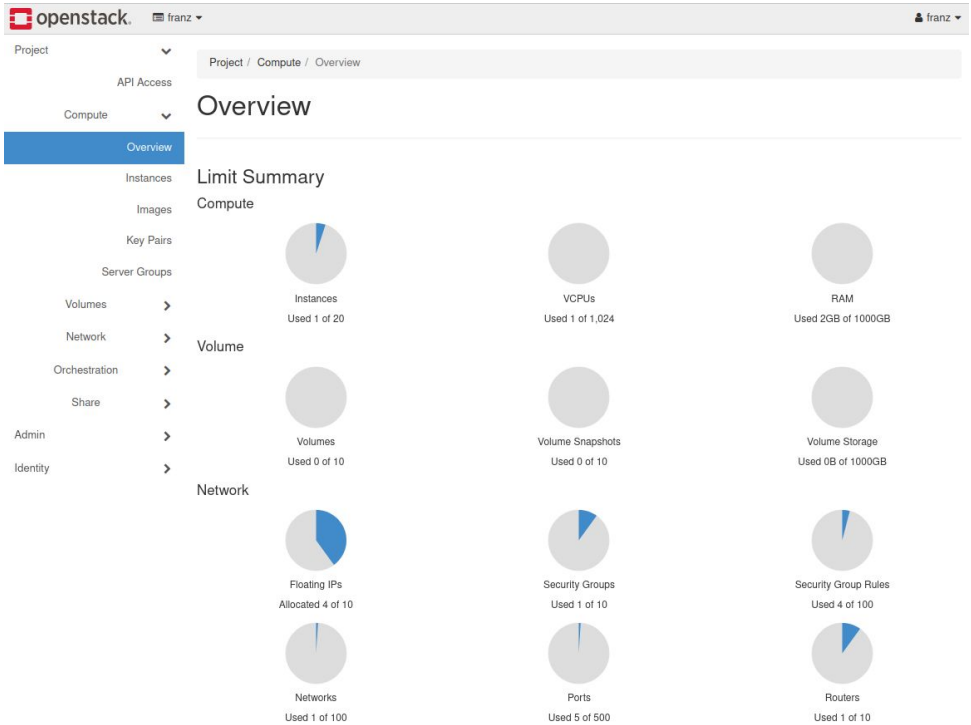
- Of course need to install CLI first
  - pip install python-openstackclient

- While you're at it, might as well install manila- and octavia-CLI as well:
  - Manila (shared storage): pip install python-manilaclient
  - Octavia (loadbalancer): pip install python-octaviaclient

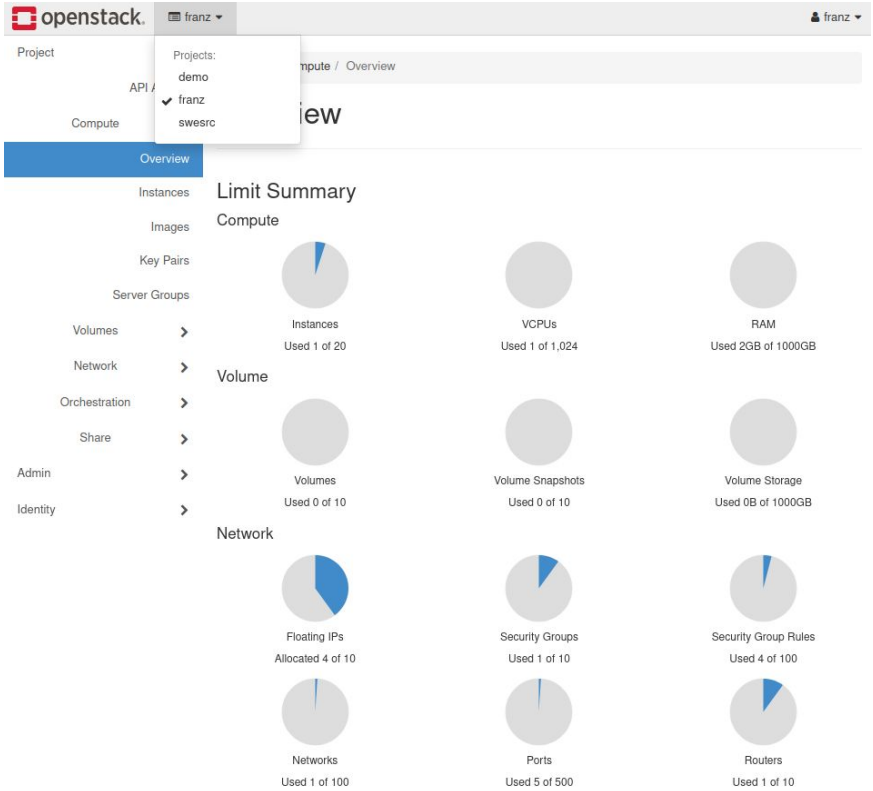
The cloud is up – first steps

- Let’s take a look at init-runonce
  - https://github.com/openstack/kolla-ansible/blob/master/tools/init-runonce
- What it does:
  - Some sanity checks (CLI installed, access credentials available…)
  - Defines some network masks for both the private and the public network
  - Creates:
    - an image (cirros)
    - a network (router, network, subnet, gateways)
    - security rules (i.e. firewall settings)
    - ssh-key pair
    - a few flavors
    - quotas for the ‘admin’ project
  - First helping hints on how to create a VM

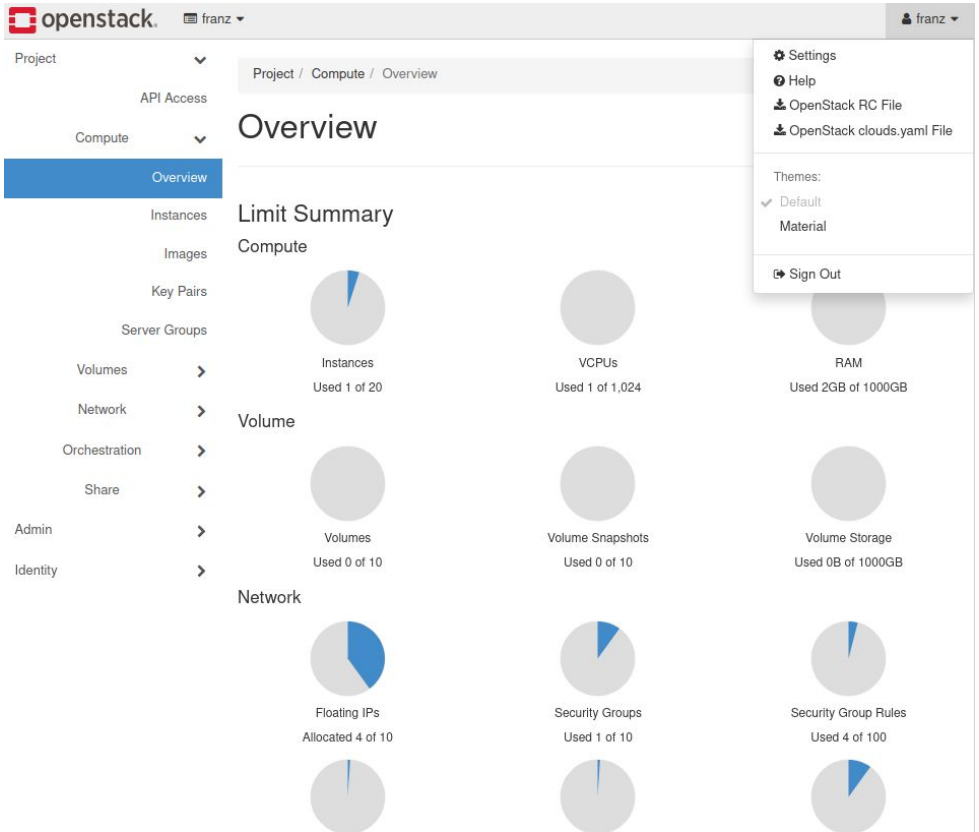
The cloud is up – check out Horizon



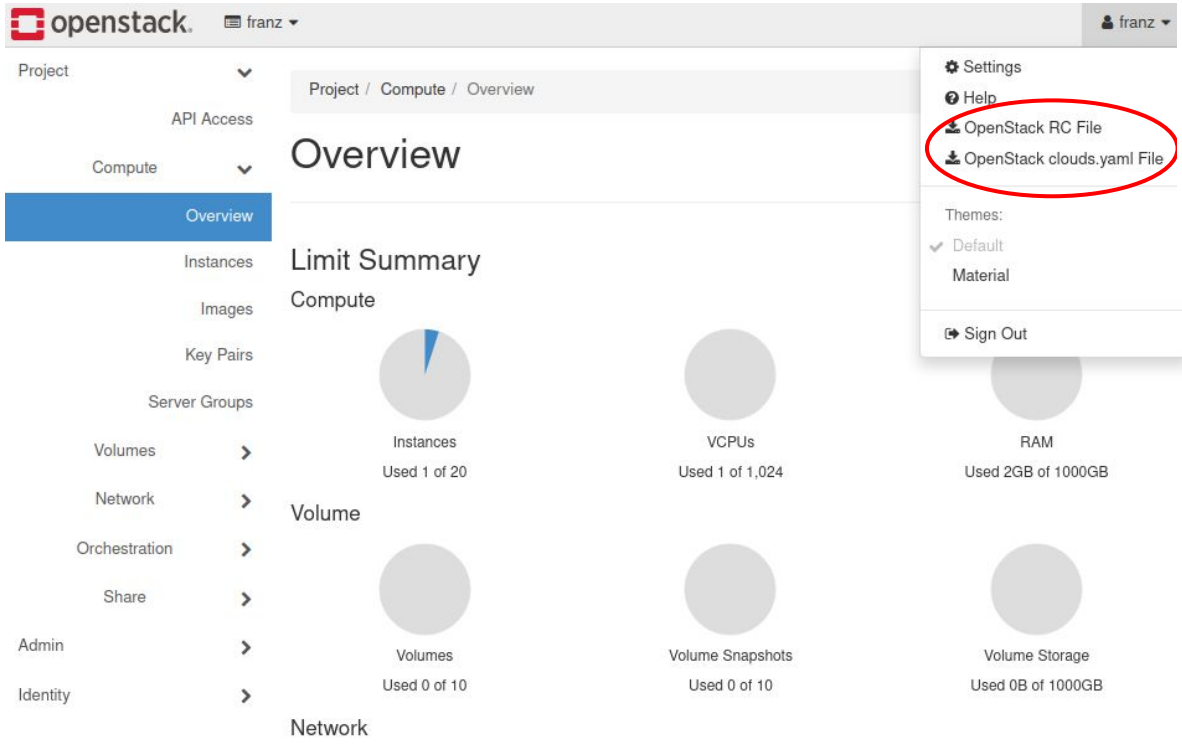
The cloud is up – check out Horizon



The cloud is up – check out Horizon

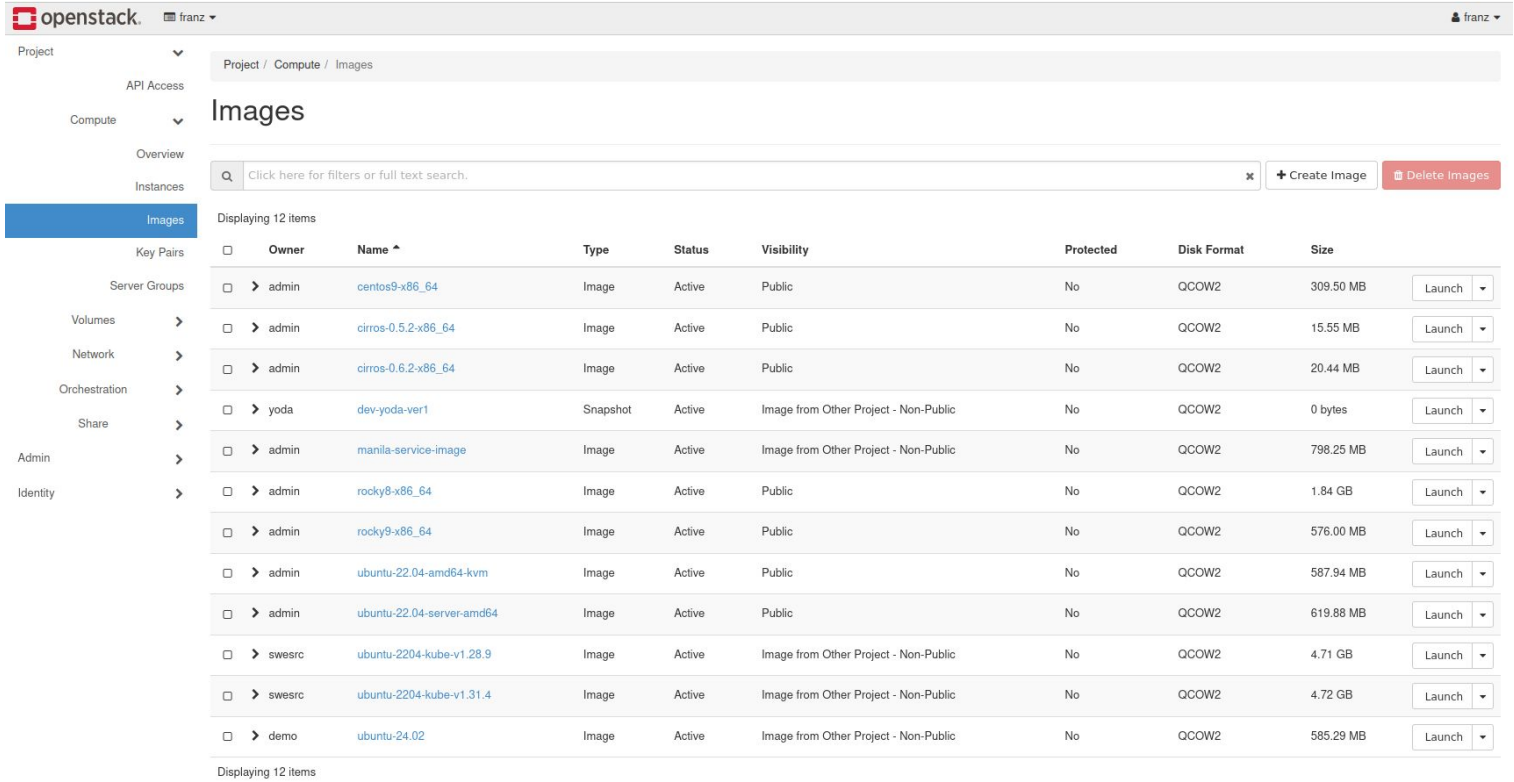


The cloud is up – check out Horizon

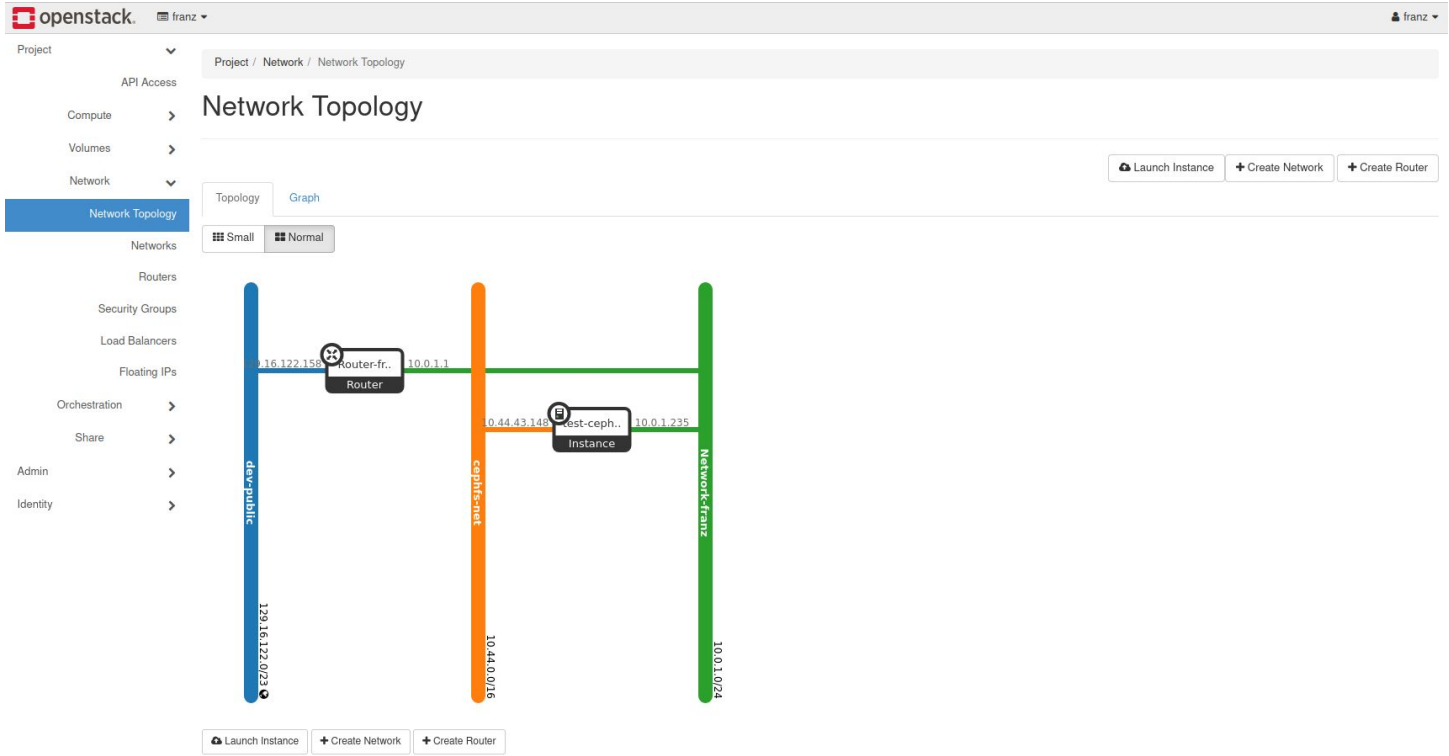


Access credentials for use with CLI client or, e.g., clusterAPI or Terraform

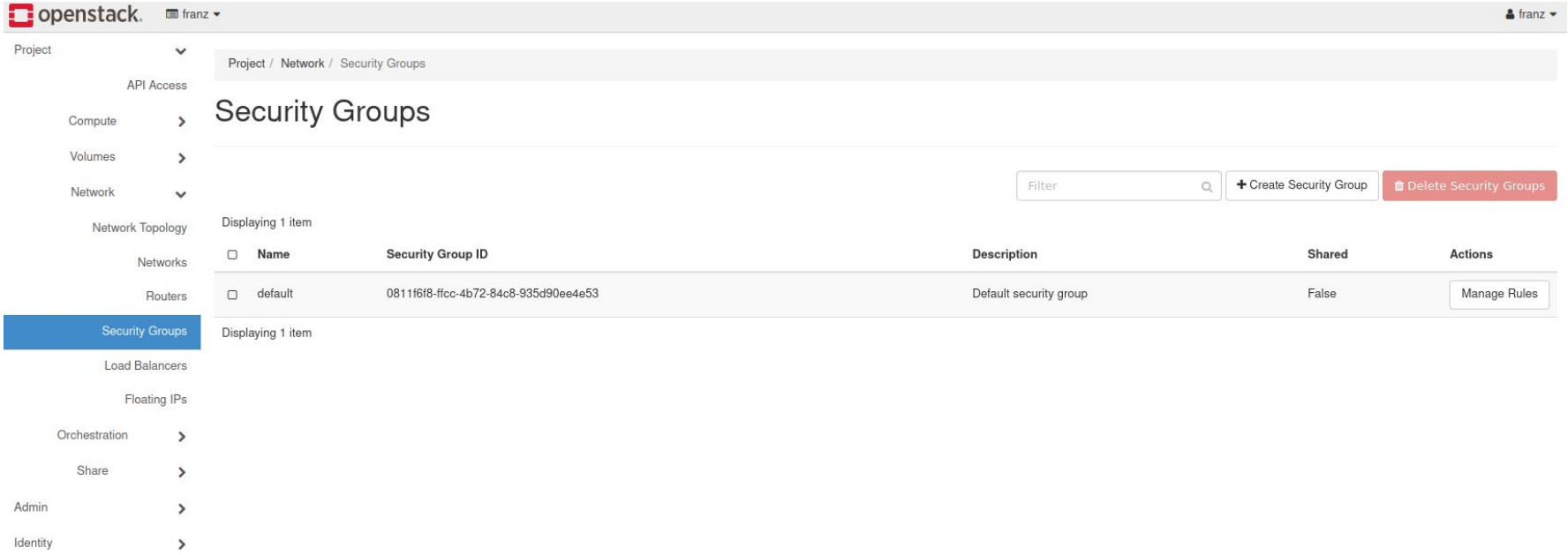
The cloud is up – check out Horizon: Images



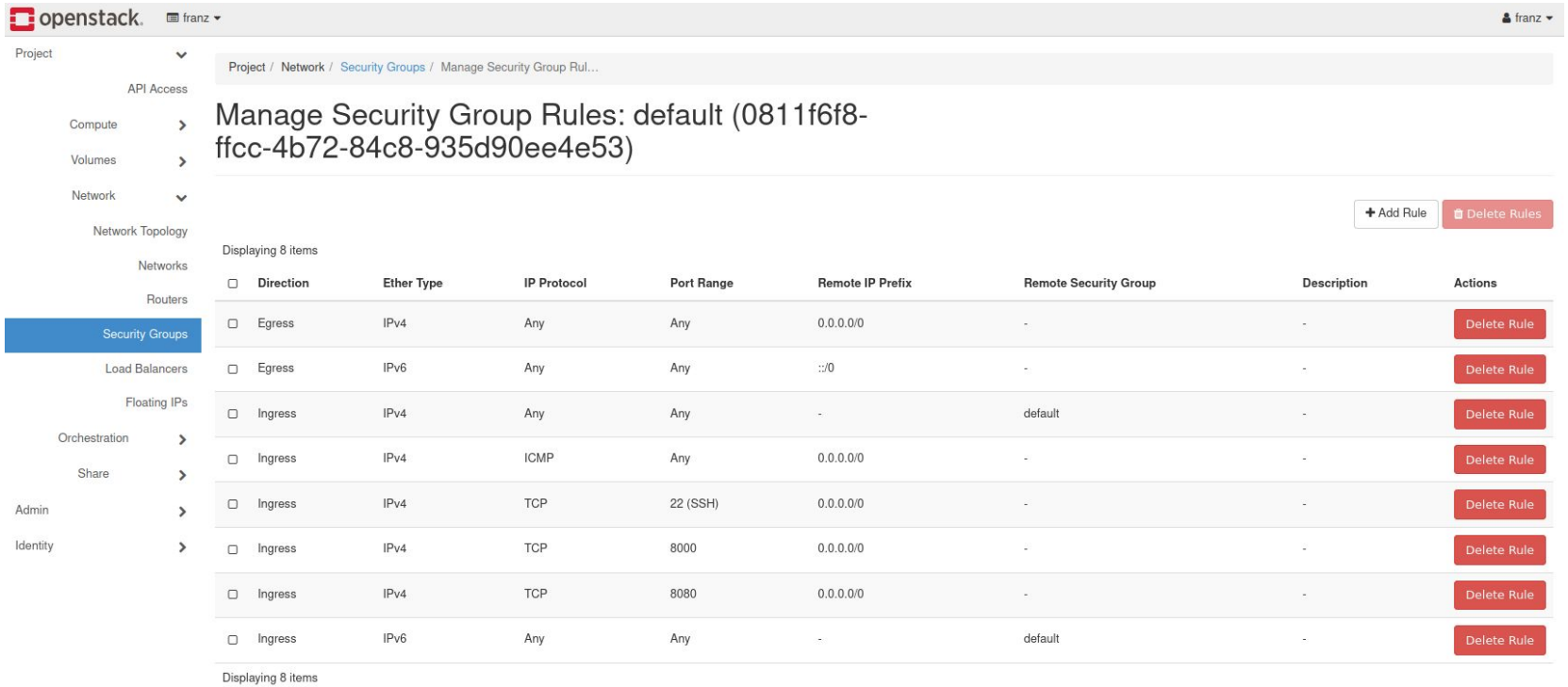
The cloud is up – check out Horizon: Networks



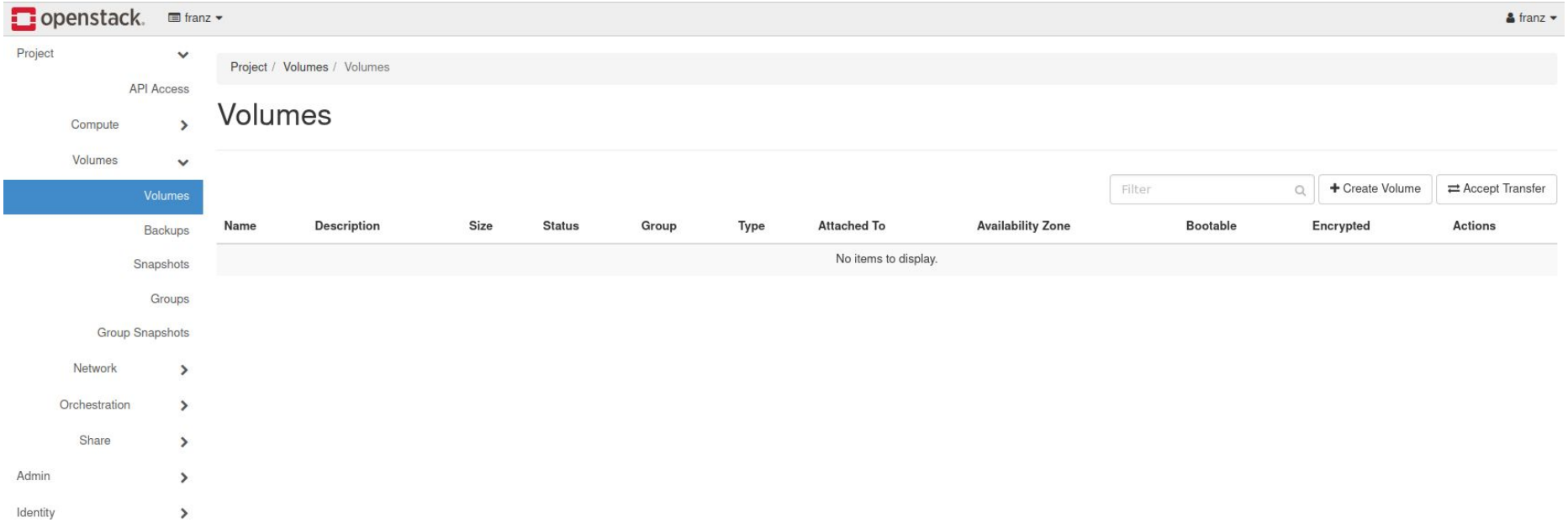
The cloud is up – check out Horizon: Security Groups



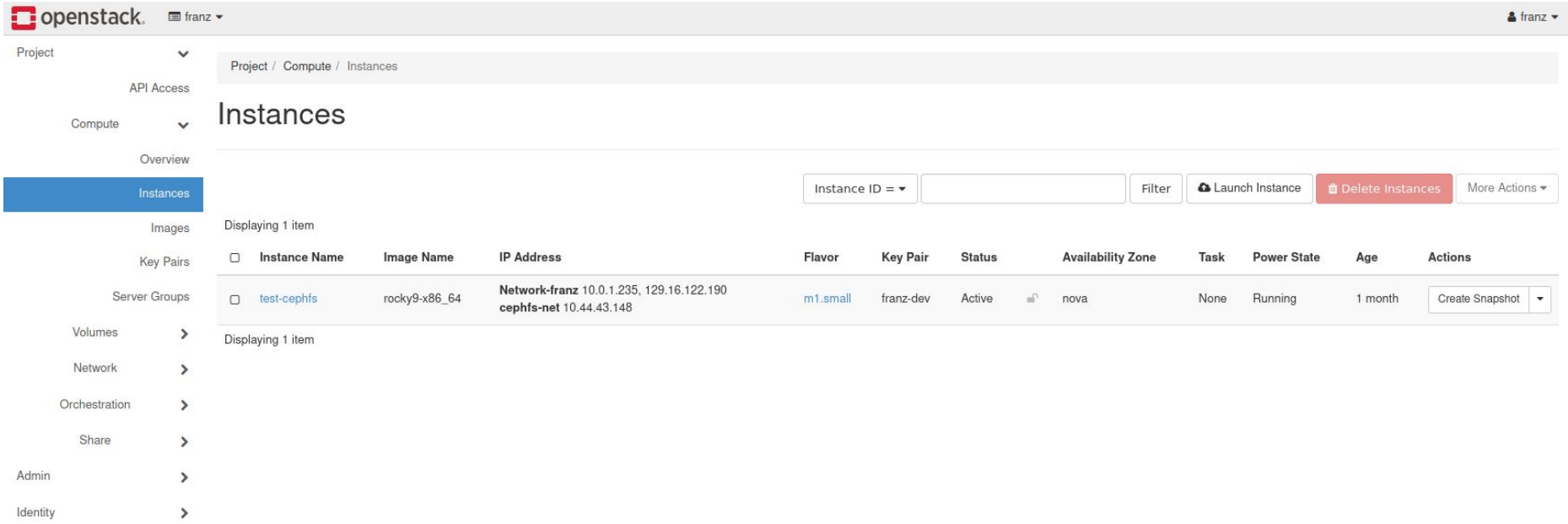
The cloud is up – check out Horizon: Security Groups



The cloud is up – check out Horizon: Volumes (storage)



The cloud is up – check out Horizon: Instances



The cloud is up – Hands-on Session!

- Pick a project/user
- Log in at dev-cloud.swesrc.chalmers.se
- Create a VM that you can log into from your laptop!

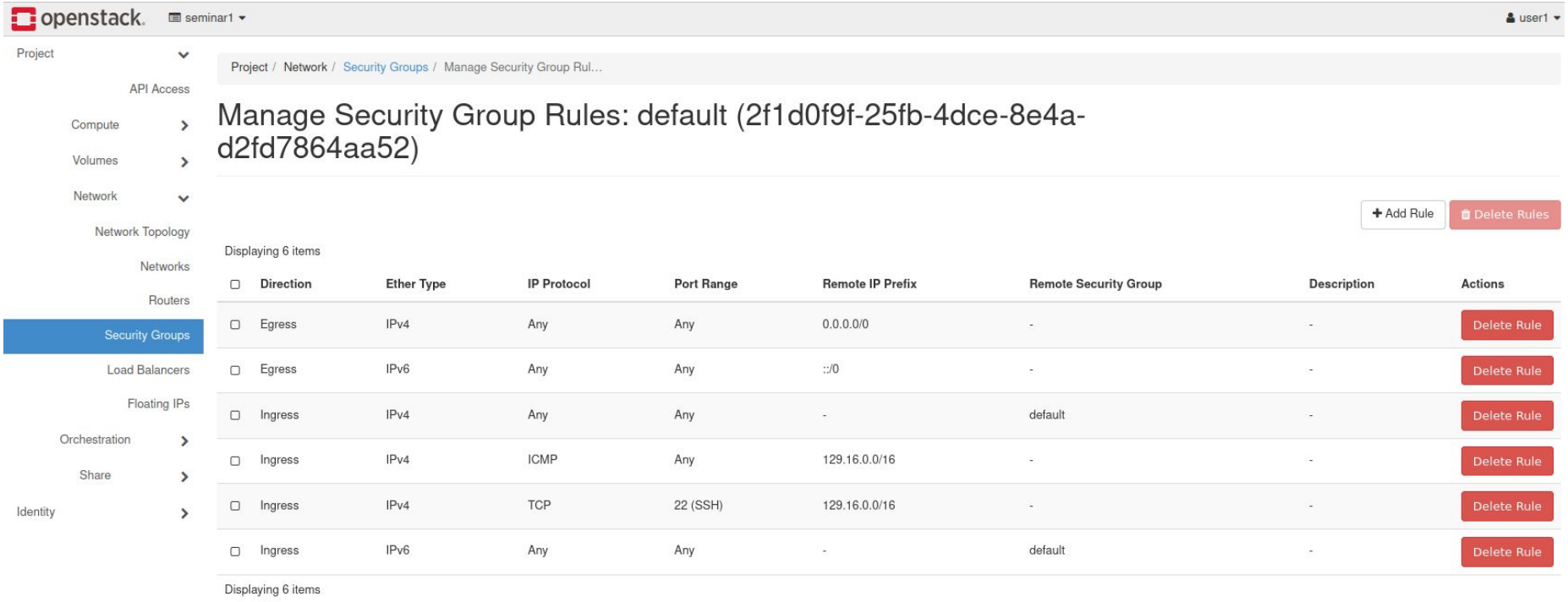
The cloud is up – Hands-on Session!: Steps

- Upload/create an ssh keypair
- Create a router (pick ‘dev-public’ as external network)
- Create a network (not shared)
  - Create a subnet (pick a gateway IP from your Network Address range)
    - E.g., if Network Address is 10.3.0.0/23, you could pick 10.3.1.1
- Add your network to the router (Routers → “Add Interface” to the router you created earlier)
- Create a VM (aka: Launch an Instance)
  - “Create New Volume” → No
  - Pick an image
  - Pick a Flavor (for rocky and ubuntu go for at least m1.medium)
  - Pick your recently created network as the network
  - “Launch Instance”
  - Once it’s up: Associate Floating IP

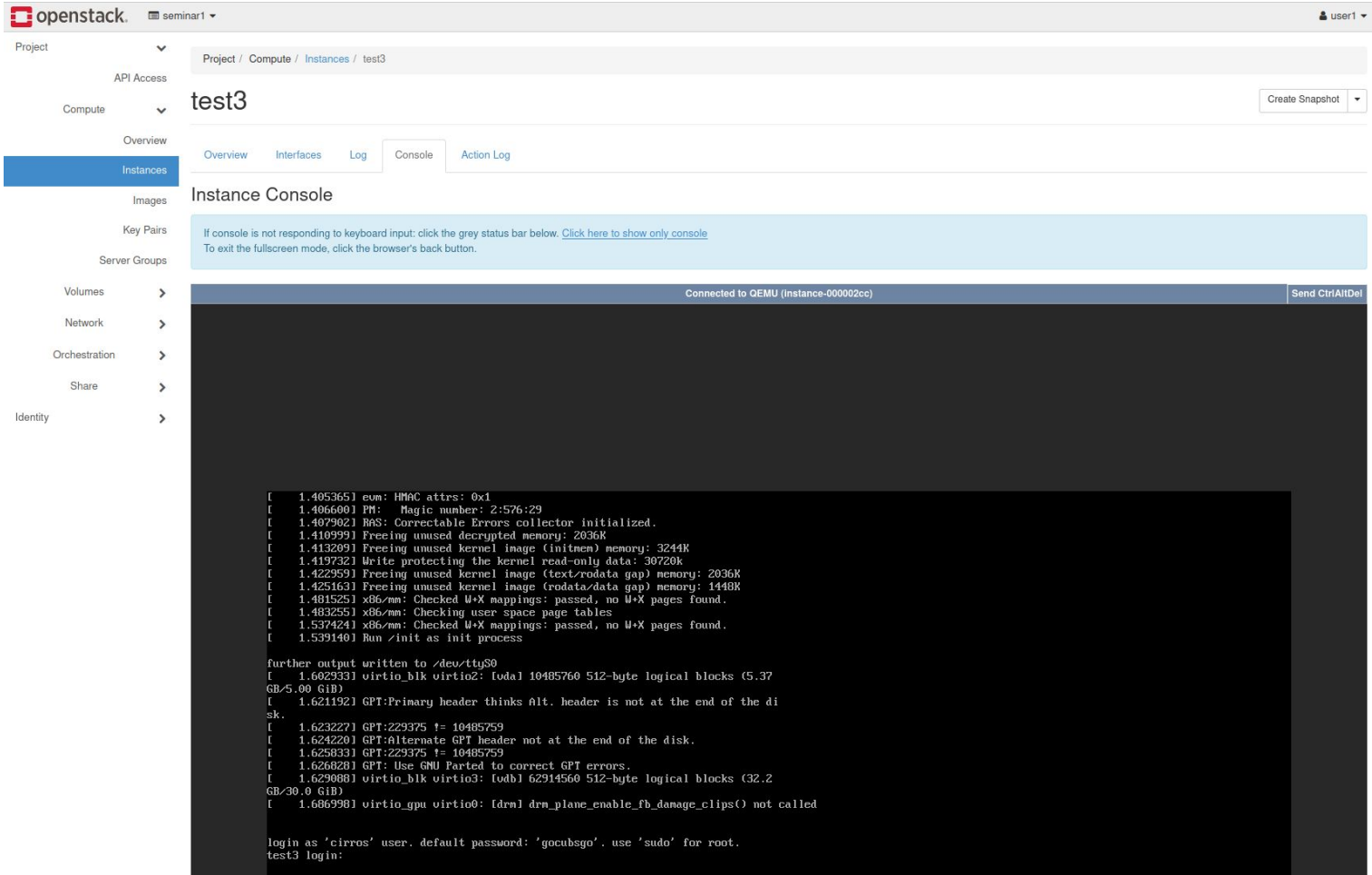
The cloud is up – Hands-on Session!: Steps

- Try and ssh into your machine with the ssh-key you added at creation
  - For rocky images: rocky@<IP>
  - For Ubuntu: ubuntu@<IP>
  - For Cirros: cloud@<IP>
- **Ensure you have the correct Security Rules in place!**
  - Make sure the security group associated with your VM has ssh-access enabled (TCP traffic on port 22; ideally limited to access from Chalmers network)

The cloud is up – Hands-on Session!: Steps



The cloud is up – Hands-on Session!: Alternative access via Horizon



The cloud is up – Hands-on Session!: Steps

- Try and ssh into your machine with the ssh-key you added at creation
  - For rocky images: rocky@<IP>
  - For Ubuntu: ubuntu@<IP>
  - For Cirros: cloud@<IP>
- Add a volume
  - Create one of size, say, 1 GB
  - Attach it to your VM
  - Once that’s done, you will still need to add a partition table and create a file system on the device

Doing the steps above via the CLI

- Create ssh-keypair:
  - openstack keypair create <key_name> [--public-key <path-to-existing-key>]
- Create a router:
  - openstack router create <router_name>
- Create a network
  - openstack network create <network_name>
- Create a subnet
  - openstack subnet create <subnet_name>
    --network <network_name>
    --subnet-range 10.10.1.0/24
    --gateway 10.10.1.1
- Add Network to the router
  - openstack router add subnet <router_name> <subnet_name>
- Set router's external network to enable floating IP access
  - openstack router set --external-gateway dev-public <router_name>

Doing the steps above via the CLI

- Create server/instance/virtual machine
  - openstack server create <server_name>
    --image <image>
    --flavor <flavor>
    --network <network_name>
    --key-name <key_name>
- Associate floating IP
  - Create a new floating IP first
    - openstack floating ip create dev-public
  - Add this floating IP to your server
    - openstack server add floating ip <server_name> <floating-ip-address>
- Fix security rules
  - Without initial specification, the default security group added to the VM will be the security group called “default”
  - But we can create new ones and add them to the server (also remove the default one if we want)

Doing the steps above via the CLI

- Fix security rules (continued…)
  - To just add ssh-access to the existing ‘default’ group:
    - openstack security group rule create default --protocol tcp --dst-port 22 --ingress --remote-ip 129.16.0.0/16
  - Create a new group, add ssh-access, remove ‘default’ from and add new group to server
    - openstack security group create <sg_name>
    - openstack security group rule create <sg_name> --protocol tcp --dst-port 22 --ingress --remote-ip 129.16.0.0/16
    - openstack server add security group <server_name> <sg_name>
    - openstack server remove security group <server_name> default
  - Names in OpenStack are not unique, i.e. there may very well exist several servers/networks/subnets/routers/volumes/security groups with the same name. In that case one needs to replace the names above by the actual ID of that particular service; this can be determined by running openstack [server, network, subnet…] list

Doing the steps above via the CLI

- Create a volume
  - openstack volume create --size <size_in_GB> <volume_name>
- Add volume to server
  - openstack server add volume <server_name> <volume_name>

Controlling where VMs are scheduled

Host aggregates

- Group hosts according to certain metadata (e.g. “gpu=true”; and/or “ssd=true”)
- Essentially helps the scheduler to place VMs on the right host
- Invisible to the user
- Same host can be in several different aggregates

Availability Zones

- A special kind of host aggregates (metadata key: availability_zone=<zone>)
- Same host can be in at most one AZ
- Visible to the user; i.e. they can choose which AZ their resources are placed in

Time for Part II*

*If time permits we can play a little more towards the end.

Projects, Users, Groups and Roles

- Managed internally by Keystone (Identity service)
- Used for authentication and authorization
- Authentication - Who is this person? Are they who they claim they are?
  - Can map OS-keystone user to an IdP such as KeyCloak or Active Directory
  - Can also create users on demand from other identity providers (federation)
- Authorization - What is this person allowed to do? Are they a regular user or an administrator? Which resources are they allowed to manage?
- Handled by reference to roles and project membership
  - Can be done by group mapping in the IdP
- Questions to be resolved:
  - Bootstrapping – How does a user log in the first time?
  - Persistence – do we create ephemeral or persistent users?

Users in OpenStack

OpenStack has a relatively minimalistic user model:

- Username (string)
  - Enforced to be unique if and only if user is persistent
- Email (optional)
- Domain
- UUID
- Description (Free text)
- Enabled (Boolean)
- Primary Project
- Type – Local (persistent) or Ephemeral. Ephemeral is the **default** when using an external IdP

We want to:

- Securely link verifiable physical-biological persons to an identifiable OpenStack user
- Allow them to log in by identifying themselves

Groups and Roles

Role

- e.g. Reader, User, Manager, Admin
- A role is set to manage a privilege but just specifies a particular “level” of privilege, rather than rich permissions
- Actual privileges bestowed through roles is defined by each service (e.g. Keystone gives certain rights to Managers for identity)
- A role is always assigned with a project link (a user is given a project-role)

Group

- Plain old list of users
- Something like Active Directory Organisational Units (OU)
- Roles can be mapped to groups and projects to automatically give users roles
- External identity providers can specify group membership

Algebra of user-role mapping

Direct assignment

USER	GROUP	ROLE	PROJECT
LEONIEL		MEMBER	PROJ1

Group-based assignment

USER	GROUP	ROLE	PROJECT
	PROJ12-MEMBERS	MEMBER	PROJ1
	PROJ12-MEMBERS	MEMBER	PROJ2
LEONIEL	PROJ12-MEMBERS		
LEONIEL		MEMBER	PROJ1
LEONIEL		MEMBER	PROJ2

Projects

- Projects own VM:s and storage containers
  - Membership of a project confers the right to access and/or manage those resources
  - Membership is equivalent to having a *role* like **member** or **admin** scoped to that project.
- Projects are themselves sparse objects
  - id, name, description, domain_id, enabled
  - options (arbitrary kv-pairs)
  - tags (arbitrary list)
  - Parents and children (projects can be nested)
- Many rich objects like resource quotas reference projects, making them a central “pivot” object that OpenStack revolves around
- An “admin” project owns all resources

Rich and sparse objects

Project (sparse)


```
{"description": "",
 "domain_id": "default",
 "enabled": true,
 "id": "b1902…",
 "is_domain": false,
 "name": "demo",
 "options": {},
 "parent_id": "default",
 "tags": []}
```


Flavor (rich)


```
{"disabled": false,
 "ephemeral": 30,
 "access_project_ids": null,
 "description": null,
 "disk": 10,
 "id": "2",
 "name": "m1.small",
 "is_public": true,
 "properties": {},
 "ram": 2048,
 "rxtx_factor": 1.0,
 "swap": 0,
 "vcpus": 1}
```


Compute quota (rich)


```
[{"Resource": "volumes",
  "Limit": 10},
  {"Resource": "snapshots",
   "Limit": 10},
  {"Resource": "gigabytes",
   "Limit": 1000},
  {"Resource": "backups",
   "Limit": 10},
  {"Resource": "groups",
   "Limit": 10},
  {"Resource": "backup-gigabytes",
   "Limit": 1000},
  {"Resource": "per-volume-gigabytes",
   "Limit": -1}]
```

Database perspective

Project - simple object (id, name, domain, etc)

User - simple object (id, name, domain, etc)

Role - simple object (id, name, domain)

Group - simple object (id, name, domain)

Assignments - foreignkey objects (User-Group, User-Role-Proj, Group-Role-Proj)

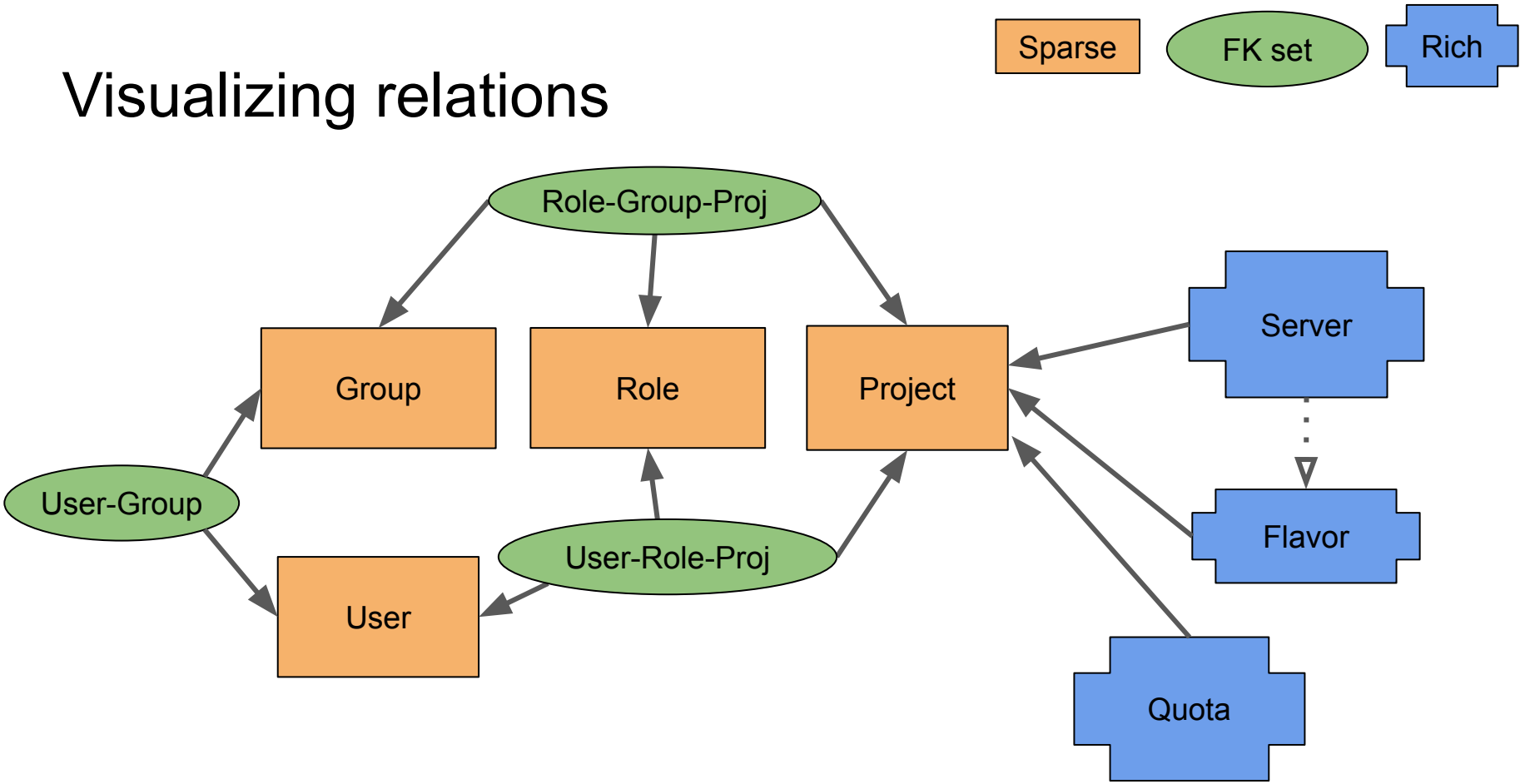
Quota - rich object (references project + quota settings like number of cores)

Flavor - rich object (has actual data, references projects that can access it)

Server - very rich object (specifies detailed VM properties, references project and user, inlines flavor information…)

Understanding this architecture is helpful when managing OpenStack!

Visualizing relations



Why SUPR?

- SUPR is already used for other services and was used for SSC
- Ensures identity of external users is verified
- Solves bootstrapping by allowing login through SUPR
- Can handle proposals, allocations and user administration
- External source of truth mitigates some risks
- Eliminates manual work in writing user-project management system
- Well-documented API for importing and exporting objects
- OpenStack limited when managing user and project metadata
- OpenStack Python bindings ⇒ implementation straightforward

Relevant objects in SUPR

- Resource - E.g. cluster, storage, cloud - Alvis, Cephyr, C3SE-Cloud.
  - Resource has Core-hours/month or GiB storage. OpenStack cloud will use Coins/month - 1 coin
- Person – Represents physical-biological person with verified identity
- Project – Represents an allocation on a set of resources and has a list of members (persons)
  - ResourceProject: Allocation on a specific resource
    - Coins for cloud, quota for storage
- Account – A ForeignKey mapping that points to a Person and a Resource, along with a small amount of metadata (username, Enabled/Disabled state)

Synchronizing with SUPR

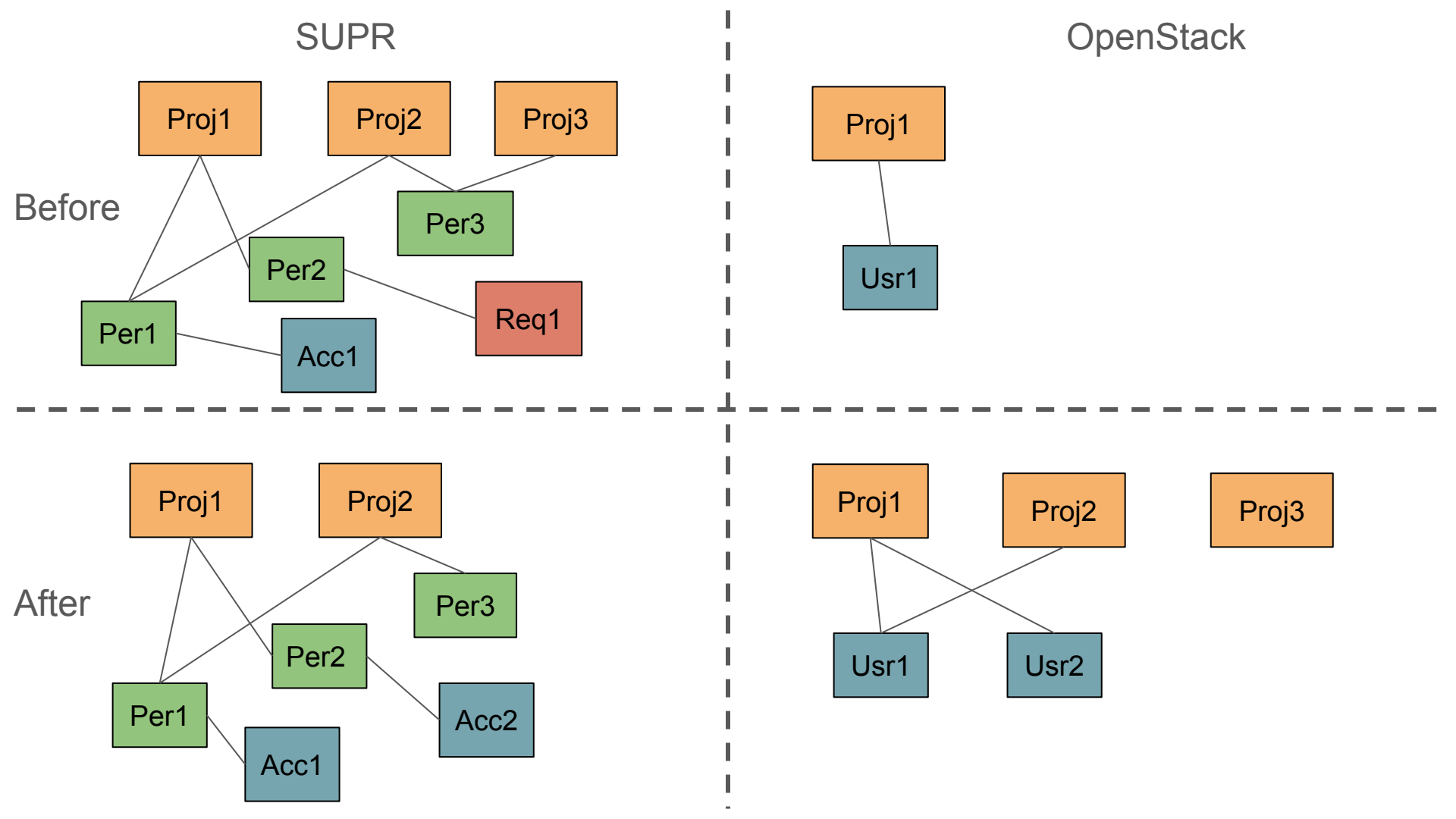
- Goal: Create SUPR-resident projects and users in OpenStack, let OpenStack create accounts in SUPR
- Should be idempotent, consistent, not lead to duplication, etc.
- We can assume that the number of users and projects will not be very large

Solution: Fetch data in a stateless manner

➔ Fetch all active projects from SUPR and create missing OpenStack projects
➔ Fetch all SUPR account requests and create users
➔ Push accounts back to SUPR
➔ Add users to projects
➔ Use SUPR accounts to authenticate

Synchronizing with SUPR

- Fetch all active projects on Cloud resource in SUPR, enumerate all projects in OpenStack
- Ensure all active projects in SUPR exist in OpenStack (match by name), otherwise create them.
- Disable OpenStack projects that have expired in SUPR
- Create/update project resource quotas. (Restrict quotas if usage exceeds quota)
- Import account requests from SUPR and create OpenStack users
- Push accounts to SUPR, deleting the account request.
- Add users to projects (through roles) in OpenStack, enable accounts if disabled.
- Disable OpenStack users which do not have active or recently expired projects.
- (Optional) Synchronize account status in OpenStack with status in SUPR
    - By not synchronizing we can modify the SUPR status to ban users


SUPR
OpenStack
Before
Proj1
Proj2
Proj3
Per3
Per2
Per1
Req1
Acc1
Proj1
Usr1
After
Proj1
Proj2
Per3
Per2
Per1
Acc2
Acc1
Proj1
Proj2
Proj3
Usr1
Usr2

OpenStack Python API

```python
import openstack

class OpenstackObjects:
    def __init__(self, cloud):
        self.cloud = cloud
        self._connection = openstack.connect(cloud)

    @property
    def connection(self):
        return self._connection

    @property
    def member(self):
        return self.connection.identity.find_role('member')

    def get_projects(self):
        return self.connection.identity.projects()

    def get_users(self):
        return self.connection.identity.users()

    def get_servers(self):
        return self.connection.compute.servers(all_projects=True)

    def get_volumes(self):
        return self.connection.volume.volumes(all_projects=True)

    def get_backups(self):
        return self.connection.volume.backups(all_projects=True)
```

```python
def set_project_storage_quota(self,
                              project_id,
                              storage_in_gb=None,
                              number_of_volumes=None,
                              number_of_snapshots=None,
                              number_of_backups=None):
    """
    Sets the 'gigabytes' storage quota which modifies the t
    of volumes and volume snapshots, as well as the number

    If any one value is not set, it is left unchanged.
    """
    kwargs = {}
    if storage_in_gb is not None:
        kwargs['gigabytes'] = storage_in_gb
    if number_of_volumes is not None:
        kwargs['volumes'] = number_of_volumes
    if number_of_snapshots is not None:
        kwargs['snapshots'] = number_of_snapshots
    if number_of_backups is not None:
        kwargs['backups'] = number_of_backups
    return self.connection.block_storage.update_quota_set(
        project_id, **kwargs)
```

Combining with the SUPR API

```python
def import_supr_projects(dry_run=False, verbose=False):
    supr = SUPR()
    # Search parameters
    params = {
        'resource_id': config['supr']['resource_id'],  # C3SE
        'end_date_ge': datetime.date.today() - datetime.timedelta(days=30),
    }
    try:
        supr_projects = supr.get('/project/search/', params=params)
        supr_resource = supr.get(f'/resource/{config["supr"]["resource_id"]}')
    except SUPRHTTPError as e:
        # We want to show the text received if we get an HTTP Error
        logger.info("HTTP error {0} from SUPR:".format(e.status_code))
        logger.info(e.text)
        raise

    if verbose:
        logger.info("Currently there are {0} active projects at C3SE present in SUPR".format(
            len(supr_projects.matches)))
    # active_project_requests = ProjectRequest.objects.all().values_list('suprid', flat=True)
    openstack_projects = {o.name: o for o in openstack_objects.get_projects()}
    for supr_project in supr_projects.matches:
        if supr_project.name in openstack_projects:
            openstack_project = openstack_projects[supr_project.name]
        else:
            openstack_project = openstack_objects.create_project(supr_project.name)
        import_project_members(supr_project, openstack_project, supr_resource, dry_run)
```

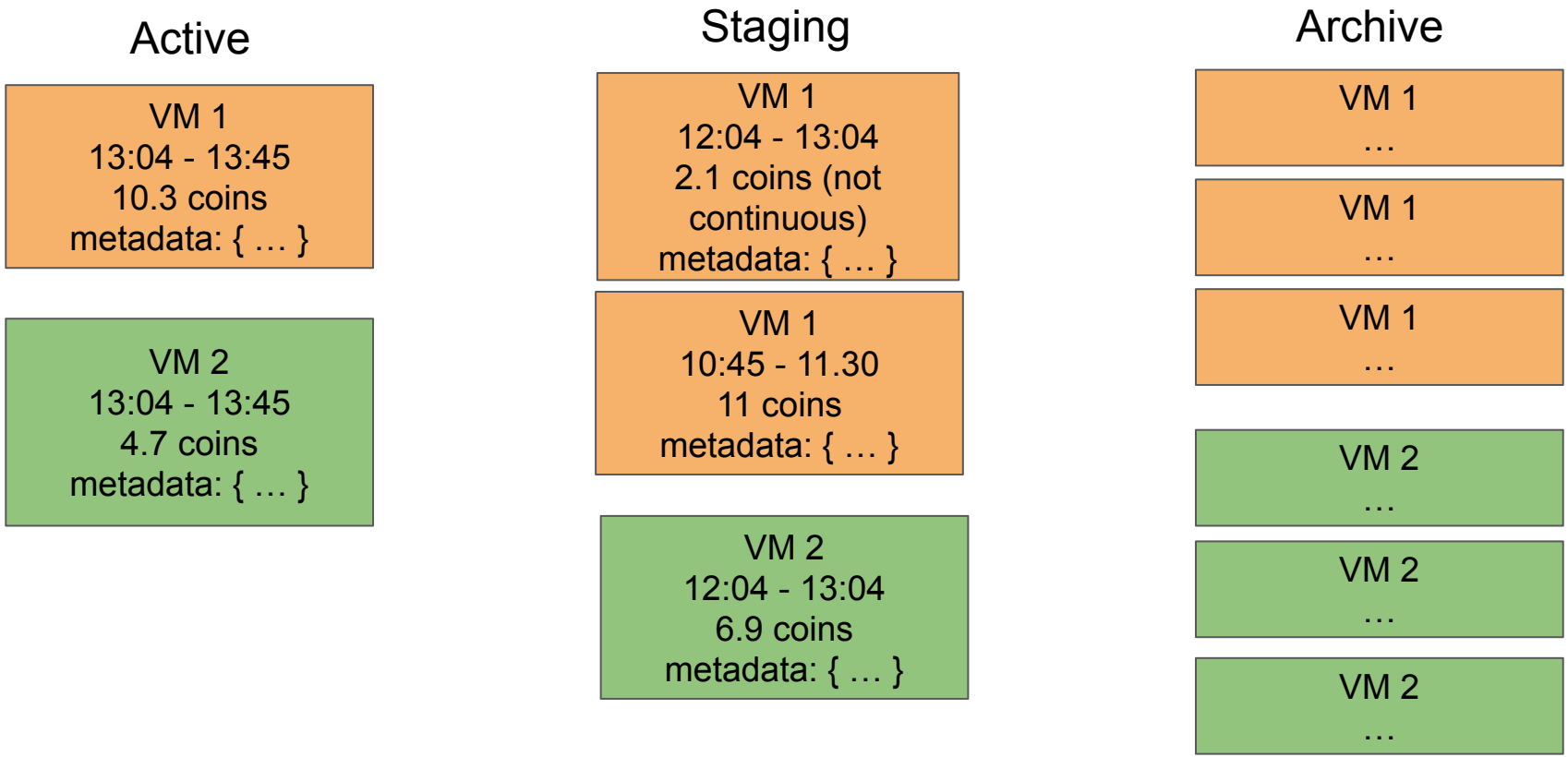
Handling account requests

- Using account requests is optional, we could just create accounts directly
- But account requests allow users to pick preferred usernames
  - Plus, we avoid creating users for people who are added to a project but never log in…
- User submits up to 3 preferred usernames
- We clean the 3 usernames (force lowercase, strip everything except [a-z0-9])
  - this way we can identify administrators by adding a prefix, since dashes will be stripped
- Check usernames against a profanity filter, discard if they fail the check
- See if remaining usernames already exist, discard if so
- If valid username remains, create user with first remaining one
- Otherwise use up to 8 letters of the user's given name, add numerical suffix until unique (leonard, leonard1, leonard2, leonard3 …)
- Could use more complicated rule like Chalmers PDB but this risks creating inappropriate-looking usernames unless we use the profanity check, and some given names can fail a profanity check (Scunthorpe problem)

Accounting

- Each flavour of VM has a cost in coins/hour
- An accounting service frequently (say once per minute) probes for running VM:s and looks up the cost of each VM, its associated project, and other metadata.
- A list of running VM:s and their cost per hour is fed into an SQL query that trapezoidally time-integrates the cost per hour into a total cost for each VM and puts that into a relation with a row for each VM and columns for the cost, integration time, and metadata.
- A second service regularly (say, once per hour) extracts chunks of time-integrated data which have been integrated over a sufficiently long time and puts in a “staging” relation for reporting.
- A third, daily service gets data and metadata from staging entries for reporting to SAMS. After reporting, data is placed into a third, archive relation.
- Utilities can be used to query locally for the usage of a project in a given time span, which can be used to impose restrictions on projects exceeding their allocations.

Accounting


Active
VM 1
13:04 - 13:45
10.3 coins
metadata: { … }
VM 2
13:04 - 13:45
4.7 coins
metadata: { … }
Staging
VM 1
12:04 - 13:04
2.1 coins (not continuous)
metadata: { … }
VM 1
10:45 - 11.30
11 coins
metadata: { … }
VM 2
12:04 - 13:04
6.9 coins
metadata: { … }
Archive
VM 1
…
VM 1
…
VM 1
…
VM 2
…
VM 2
…
VM 2
…

Keycloak, SUPR and OpenID Connect

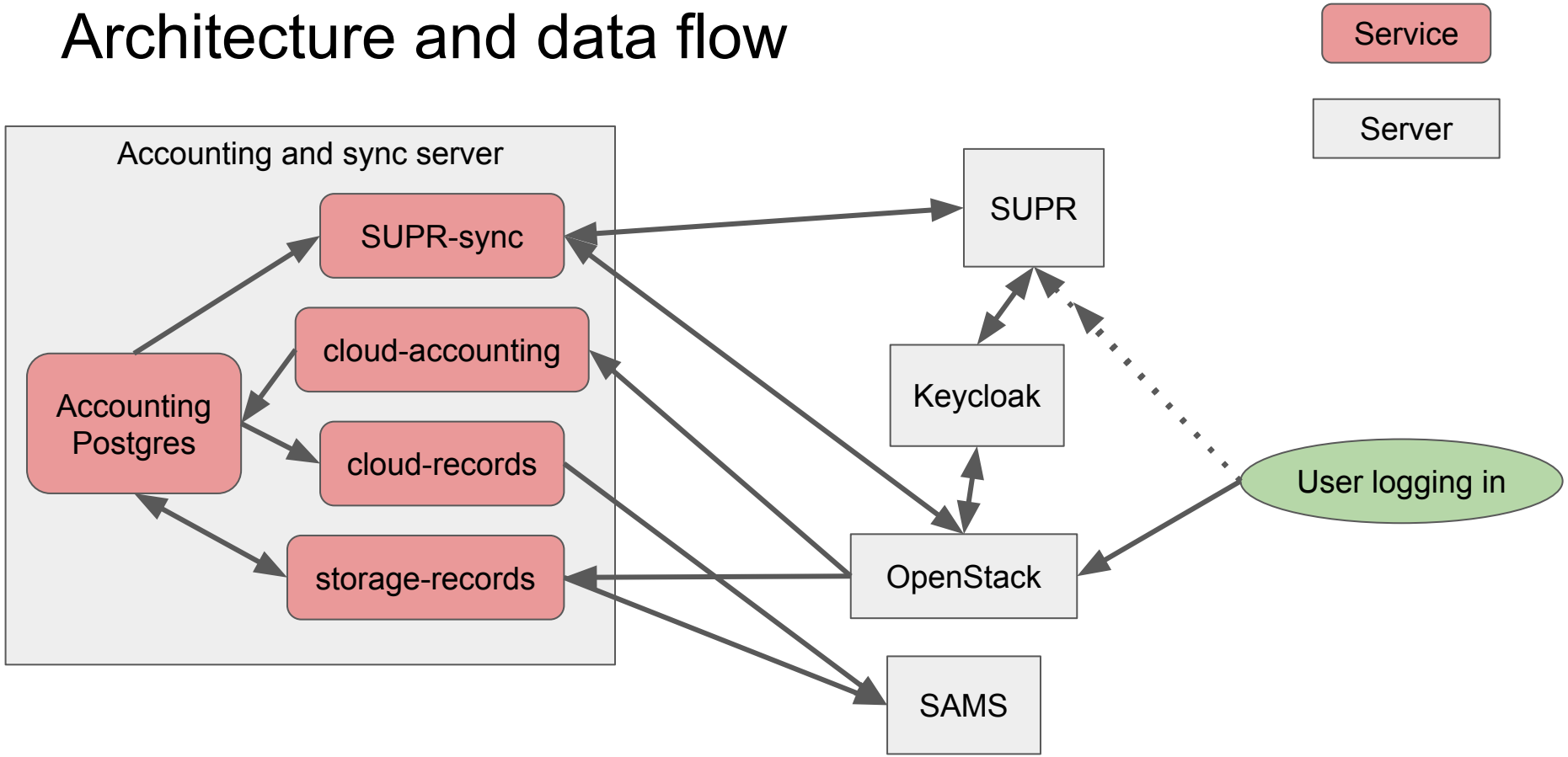
- Goal: Let users log in in a secure and verified manner
  - Use authentication against SUPR exposed over OpenID Connect
- Keycloak uses the SUPR Authentication API to let a user log in through SUPR.
- On successful login, Keycloak examines user metadata whether a user has an Account on a particular resource which is enabled.
- An ephemeral Keycloak user is created with the Account username that was originally pushed from OpenStack.
- User information is forwarded to OpenStack which matches the identifying user against the OpenStack user and logs them in.

keycloak_mapping.json

Other approaches

- We could get active projects from SUPR and create an OpenID Connect claim from those
- The claim could be mapped to groups in OpenStack which automatically enroll users in projects
- But - no real advantage since we anyway need to set up projects and be able to enable and disable accounts in SUPR
- However group claims could be used managing administrative users with varying levels of privilege through Keycloak tooling

Architecture and data flow



Back to Part I – more play time!*

*If time permits and everyone is keen.

Just a quick word on quotas

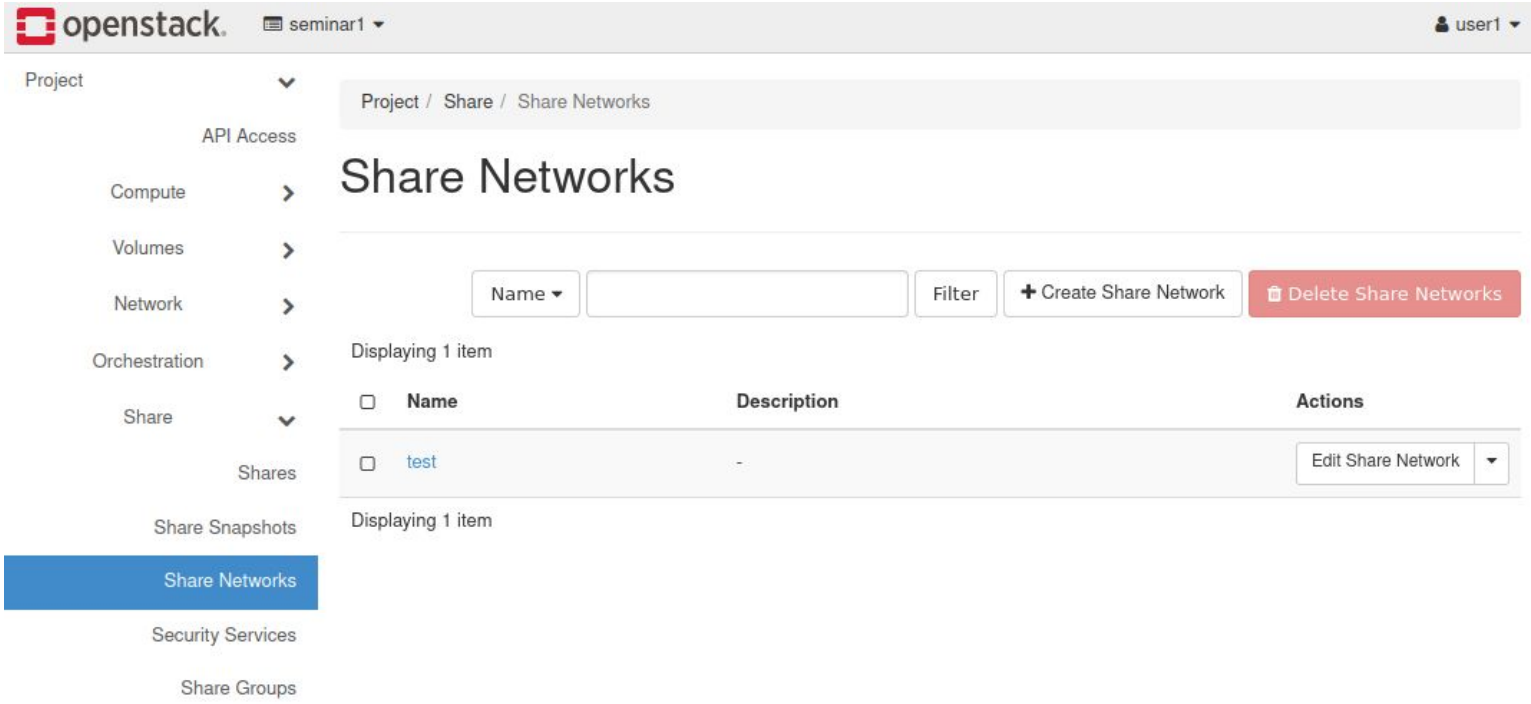
- There are default quotas set per project for pretty much everything:
  - Number of CPUs, routers, floating IPs, security groups…
  - Maximum size of volumes
  - Total maximum of allowed storage
  - Total maximum of allowed **shared** storage (yes it's counted separately)
- As admin these can be set per project
  - openstack quota set --gigabytes 1024
  - openstack share quota set --gigabytes 1024

Let’s create shared storage

- First we need to create a Share Network, i.e. a network through which VMs can access common storage

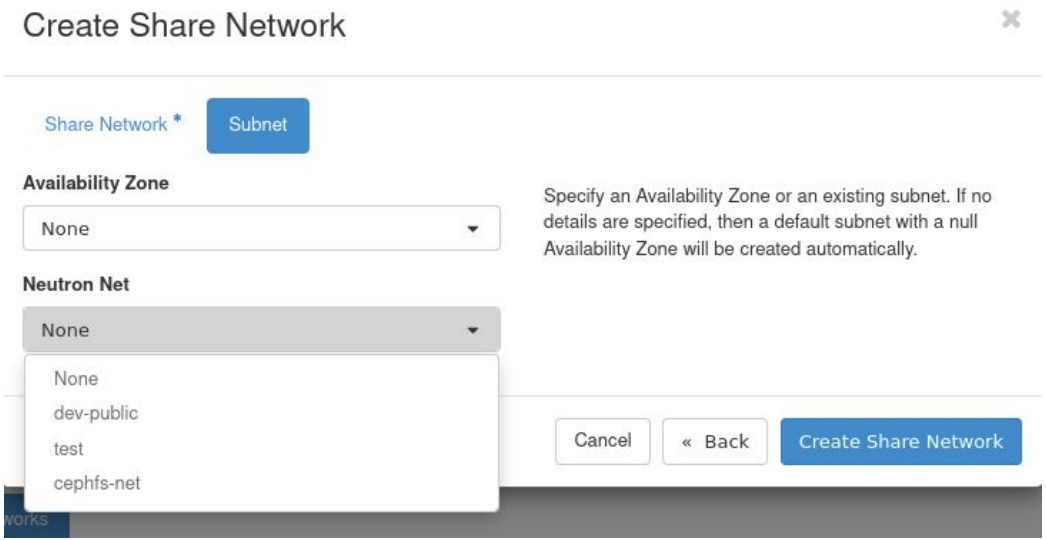
Let’s create shared storage

- First we need to create a Share Network, i.e. a network through which VMs can access common storage



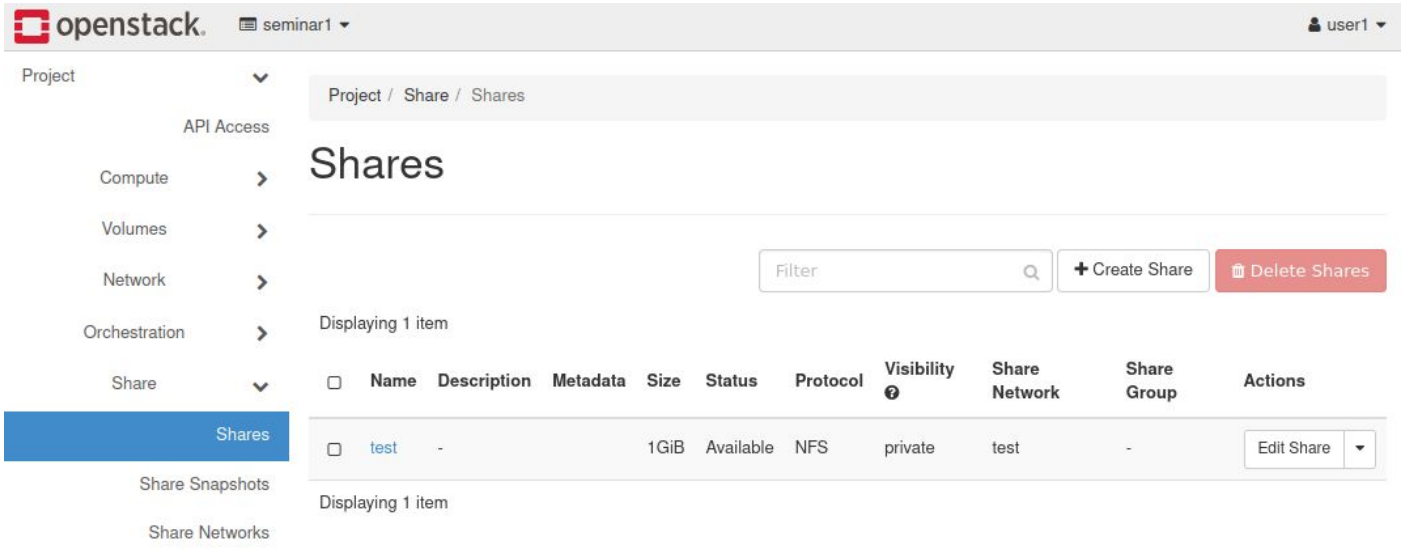
Let’s create shared storage

- First we need to create a Share Network, i.e. a network through which VMs can access common storage
- Pick as “neutron subnet” the subnet that you created for your VM(s)



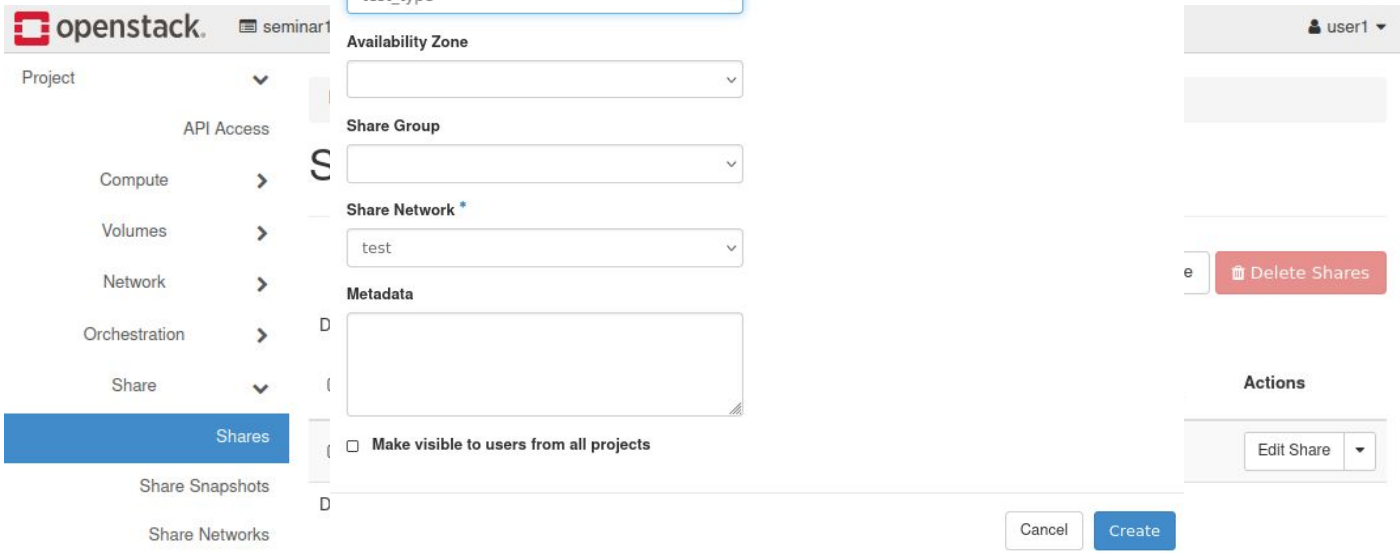
Let’s create shared storage

- First we need to create a Share Network, i.e. a network through which VMs can access common storage
- Pick as “neutron subnet” the subnet that you created for your VM(s)
- Create a share
  - Pick as share type “test_type”, protocol: “nfs”, and your share network



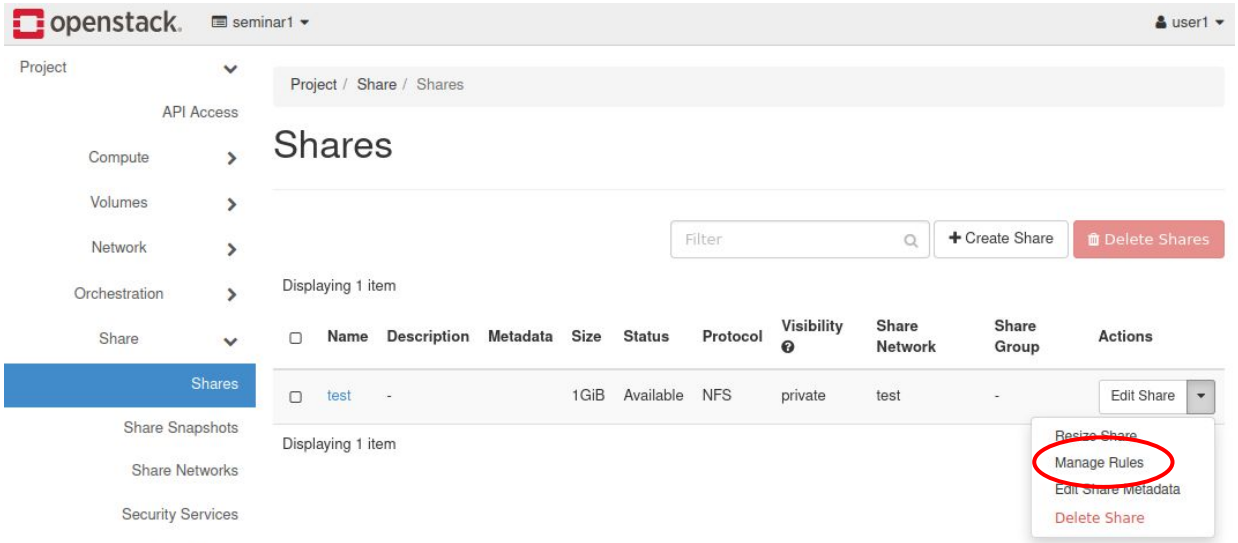
Let's create sha

- First we nee ork through which VMs can access
- Pick as "neu for your VM(s)
- Create a sha
  - Pick as nd your share network



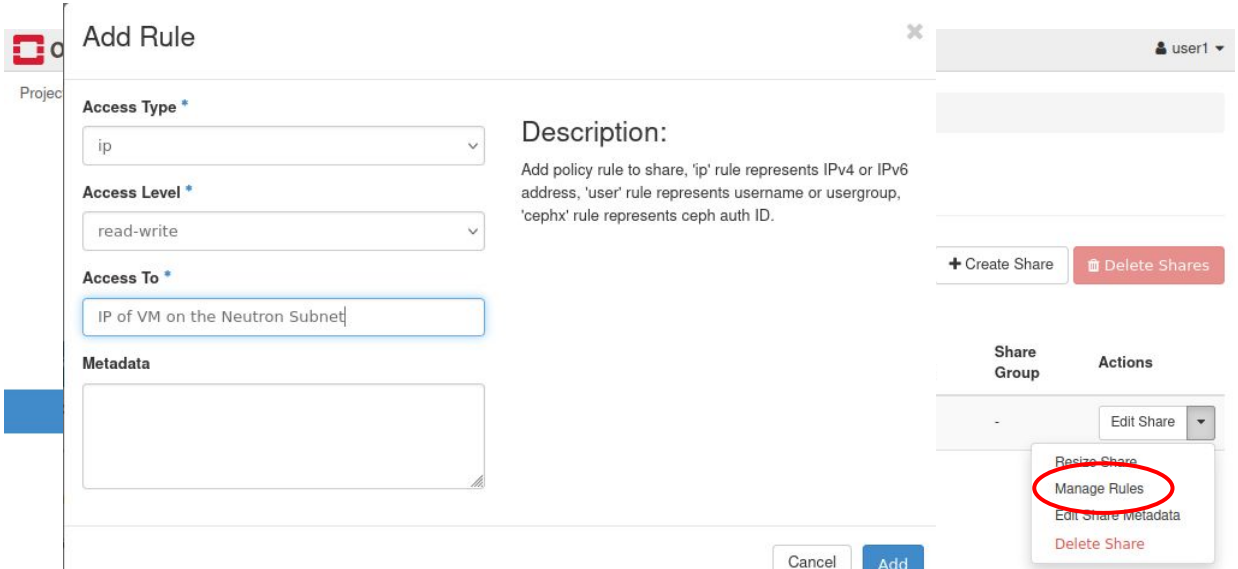
Let’s create shared storage

- First we need to create a Share Network, i.e. a network through which VMs can access common storage
- Pick as “neutron subnet” the subnet that you created for your VM(s)
- Create a share
  - Pick as share type “test_type”, protocol: “nfs”, and your share network
- Add acces rules; i.e. control which VM can access the share



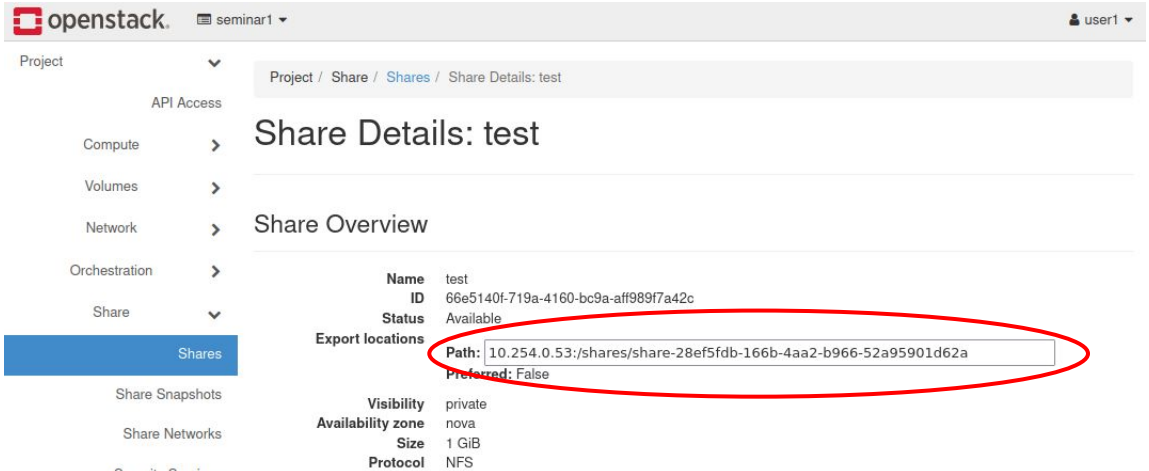
Let’s create shared storage

- First we need to create a Share Network, i.e. a network through which VMs can access common storage
- Pick as “neutron subnet” the subnet that you created for your VM(s)
- Create a share
  - Pick as share type “test_type”, protocol: “nfs”, and your share network
- Add acces rules; i.e. control which VM can access the share



Let’s create shared storage

- First we need to create a Share Network, i.e. a network through which VMs can access common storage
- Pick as “neutron subnet” the subnet that you created for your VM(s)
- Create a share
  - Pick as share type “test_type”, protocol: “nfs”, and your share network
- Add acces rules; i.e. control which VM can access the share
- Figure out the IP and path of the share



Let’s create shared storage

- First we need to create a Share Network, i.e. a network through which VMs can access common storage
- Pick as “neutron subnet” the subnet that you created for your VM(s)
- Create a share
  - Pick as share type “test_type”, protocol: “nfs”, and your share network
- Add acces rules; i.e. control which VM can access the share
- Figure out the IP and path of the share
- On your VM, assuming you have “nfs-utils” installed, you can now mount the share using that path into a directory of your choosing.

Let’s create shared storage – via the CLI

- Create the share network
  - openstack share network create --name <share_network_name>
    --neutron-subnet-id <subnet_id>
- Create the share
  - openstack share create <share_protocol (e.g. nfs)> <size (in GB)>
    --name <share_name>
    --share-network <share_network_name/ID>
    --share-type <type (e.g. test_type)>
- Add access rules
  - openstack share access create <share_name/id> ip <ip_address>
- Figure out the IP and path to the share via
  - openstack share show <share_name/id>
- Mount the share in your VM using that IP and Path