Contents

CH1: Introduction:

Traditional Computing Model …………………………………………………… 7

- One App/Server ……………………………………………………………… 7
- Traditional Switching ………………………………………………………… 7
- Drawbacks ……………………………………………………………………… 8

Cloud Computing Model ………………………………………………………… 8

- Virtual Resources …………………………………………………………….. 8
- Advantages …………………………………………………………………… 8
- Drawbacks ……………………………………………………………………… 8

Our Platform …………………………………………………………………... 9

- What Kloudak Offers ………………………………………………………… 9

……………………………………………………………………………………………..

CH2: Kloudak High-Level Architecture:

Components & Roles ……………………………………………………….. 10

- Data Services ………………………………………………………………..... 10
- Automation Services ………………………………………………………… 10
- Infrastructure ………………………………………………………………… 10

Basic Concepts ……………………………………………………………… 12

- Areas ………………………………………………………………………… 12
- Storage Pools ………………………………………………………………... 12
- JSON-WEB-TOKEN ………………………………………………………... 12

……………………………………………………………………………………………………..

CH3: Authentication & Authorization:

End-Users Security ………………………………………………………………….. 13

- Login ………………………………………………………………………………. 13
- Acquiring Token …………………………………………………………………… 13
- Token Information ………………………………………………………………. 13

Administrators Security ..…………………………………………………………... 14

- Superuser Account ………………………………………………………………. 14

……………………………………………………………………………………………………..

CH4: Deployment Guide:

Infrastructure …………………………………………………………………….. 16

- Storage (NFS Server) ……………………………………………………………. 17
- Hypervisors (KVM) ……………………………………………………………... 18
- Network (Open vSwitch) ………………………………………………………... 18
- Configuring Pools ……………………………………………………………….. 19

Middle-wares ……………………………………………………………………….. 20

- RabbitMQ ……………………………………………………………………….. 20
- Redis ……………………………………………………………………………….. 20
- Zookeeper ……………………………………………………………………….. 21

Backend Services ………………………………………………………………. 22

- Database (PostgreSQL) ………………………………………………………. 23
- Inventory ………………………………………………………………………... 23
- Controller ……………………………………………………………………….. 25

- Notification ………………………………………………………………….. 27

Automation Services ……………………………………………………….. **29**
- Monitoring ………………………………………………………………... 30
- Compute ……………………………………………………………………... 33
- Network …………………………………………………………………… 35

……………………………………………………………………………………..

CH5: End-User Guide:

Using the Dashboard ……………………………………………………….. **40**
- Login/Signup ………………………………………………………………… 40
- Workspaces ………………………………………………………………... 40
- Users …………………………………………………………………………. 41
- Networks …………………………………………………………………….. 42
- Virtual Machines ………………………………………………………….. 42

……………………………………………………………………………………..

CH6: Inventory:

Roles ……………………………………………………………………….. **44**
- User Login/Sign-up ……………………………………………………… 44
- Generating Tokens ……………………………………………………... 44
- Data Storage ………………………………………………………………. 44

ERD ……………………………………………………………………………….. **46**

Class Diagram …………………………………………………………………... **47**

……………………………………………………………………………………..

CH7: Controller:

Roles 48

- Object Manipulation 48
- Queue Monitoring 48
- Retry & Rollback 49

Components 49

- ControllerAPI 49
- Network Notification Consumer 49
- VM Notification Consumer 49

ERD 50

Class Diagram 50

....

CH8: Notification:

Roles 51

- Workspace Notification Rooms 51
- Handling End-User Connections 51
- Handling Controller Connections 51

....

CH9: Monitoring:

Roles 52

- Monitoring Hosts & Pools 52
- Collecting Resource Usage Data 52

Components ……………………………………………………………….. 52
- Celery Workers …………………………………………………………… 52
- Zookeeper Clients ………………………………………………………… 55
- RPC Servers ……………………………………………………………….. 55
…………………………………………………………………………………………..

CH10: Compute:

Roles ……………………………………………………………………… 58
- VM Creation & Deletion ……………………………………………….. 58
- Generating Network Tasks ……………………………………………… 60

Components ………………………………………………………………... 60
- VM Worker ………………………………………………………………… 60
- Host Failure Worker ……………………………………………………... 62
- Rollback Worker …………………………………………………………… 62

Class Diagram …………………………………………………………… 64

Package Diagram ……………………………………………………….. 65
…………………………………………………………………………………………..

CH11: Network:

Roles ……………………………………………………………………… 66
- Network Creation & Deletion ………………………………………... 66
- Managing VM Interfaces ………………………………………………….. 66

Components ……………………………………………………………….. 67
- Network Worker …………………………………………………………….. 67
- Rollback Worker …………………………………………………………….. 67

Class Diagram

………..

CH12: Availability, Reliability & Coordination:

Availability ………………………………………………………………………. 69

- Multiple Instances ……………………………………………………………... 69

- Infrastructure Monitoring & Recovery ……………………………………... 70

Reliability ……………………………………………………………………….. 70

- Handling Connection Failures ……………………………………………… 70

- Handling Service Failures …………………………………………………... 70

- Handling Execution Errors & Rollbacks ……………………………………. 70

Coordination …………………………………………………………………… 72

- Monitoring Coordination ……………………………………………………… 72

………..

Appendix: System APIs:

Inventory API

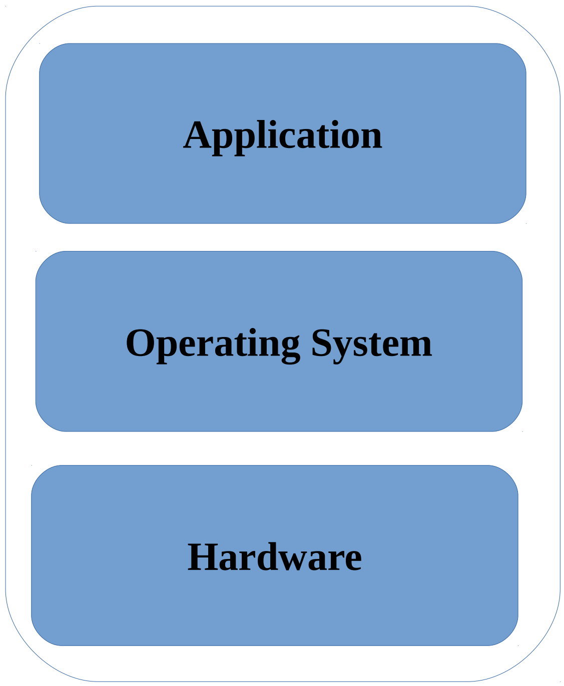
Controller API

………..

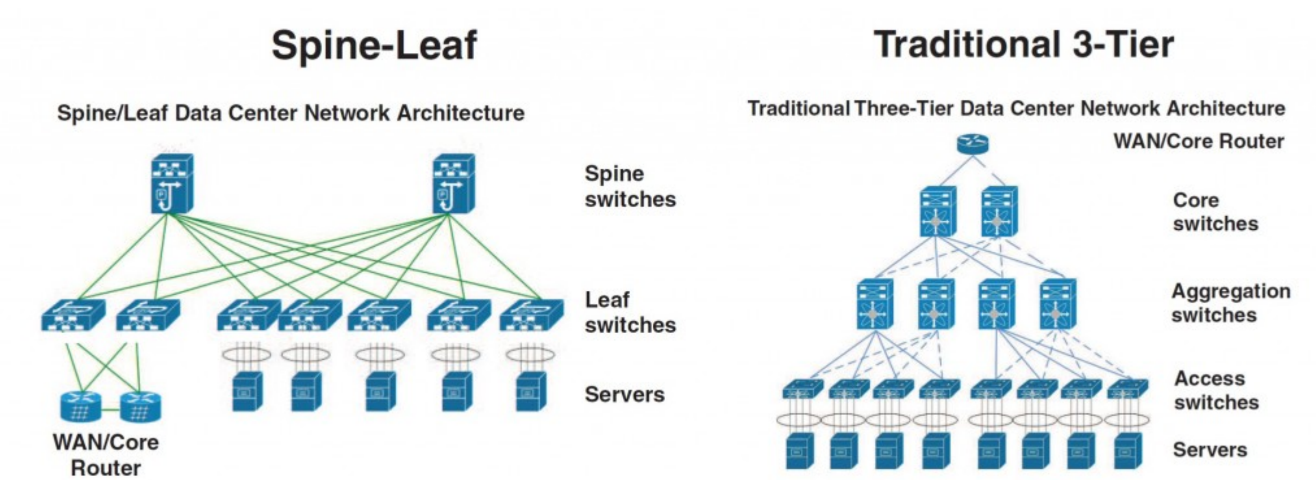
1- Introduction

Traditional Computing Model:

Traditional computing refers to using seperate physical servers for each application or service. As for networking, this models uses the traditional 3-Tiers or Spine-Leaf network architecture.



Figure(1.A) App/Server



Figure(1.B) Traditional Network

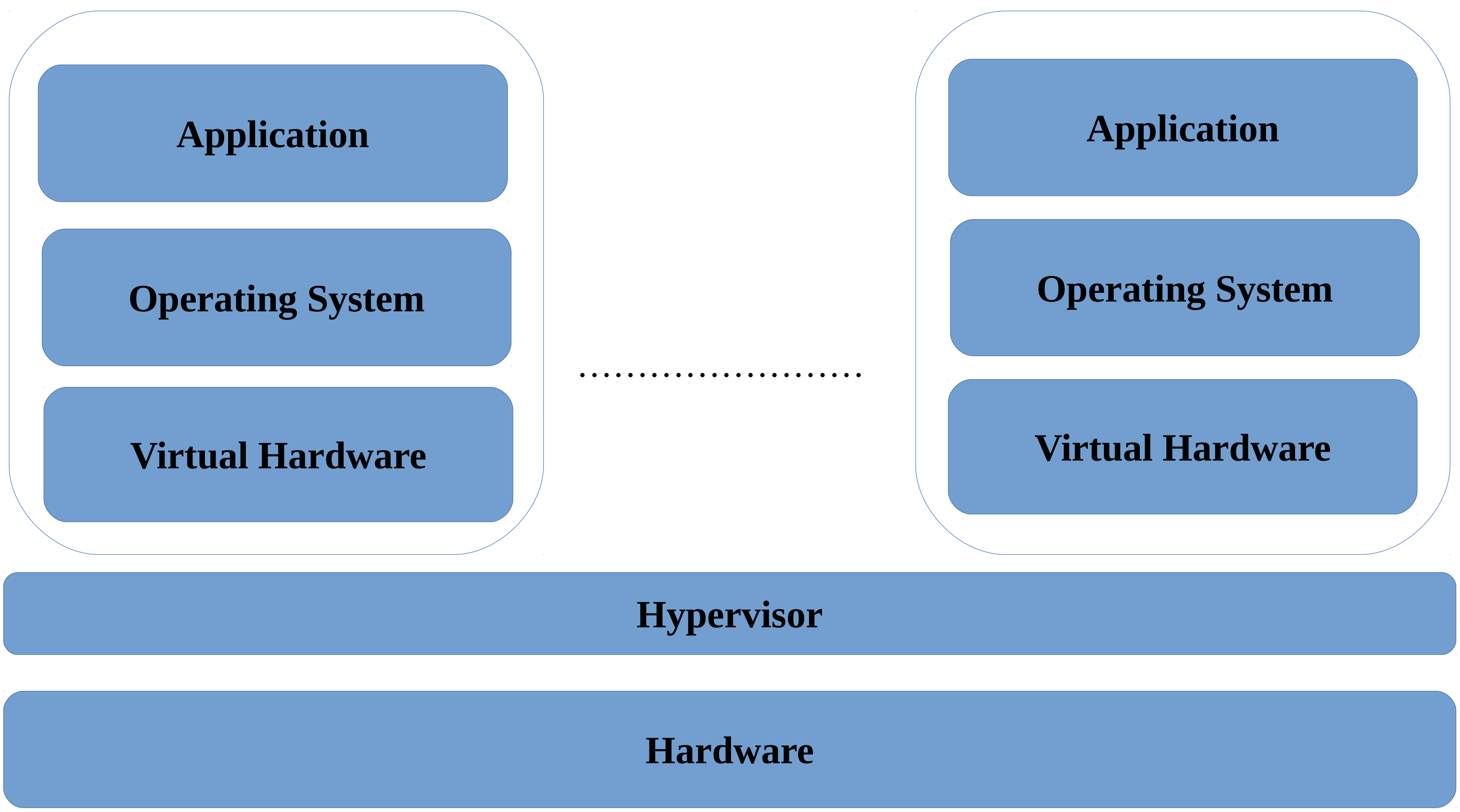
Drawbacks:

As one would expect, this model costs heavily and hard to manage and maintain. Not to mention the required time to add a new service or scale an existing one.

Cloud Computing Model:

The main goal of cloud computing is to add flexibility and self-service to infrastructure management which became possible thanks to virtualization technology. Nowadays, severs, networks and storage can be virtualized and run completely in software. Running in software means it is faster and easier to create and manipulate virtual resources using graphical dashboards, CLI tools or APIs which is provided by virtualization. Cloud adds a layer of automation and orchestration to virtualization and some form of user permission management.

Of course this model does have drawbacks especially in public cloud where security is basically zero (storing all your data at cloud providers). Also, there’s a resource usage overhead by hypervisors and software used to manage the infrastructure.



Figure(1.C) Virtualization Model

Our Platform:

Kloudak is a simple cloud platform created to provide a scalable, easy-to-implement management and automation platform for Linux-based infrastructure. Kloudak offers the basic services of users and group management as well as managing the lifecycle of virtual compute and network resources.

2- Kloudak High-Level Architecture

Components & Roles:

Kloudak components can be organized in three main groups:

1- Data Services:

Or back-end services, they are responsible for handling end-user interactions, generating and storing their data and finally sending notifications to end-users.

They are: ***Inventory, Controller, Notification and Dashboard***

2- Automation Services:

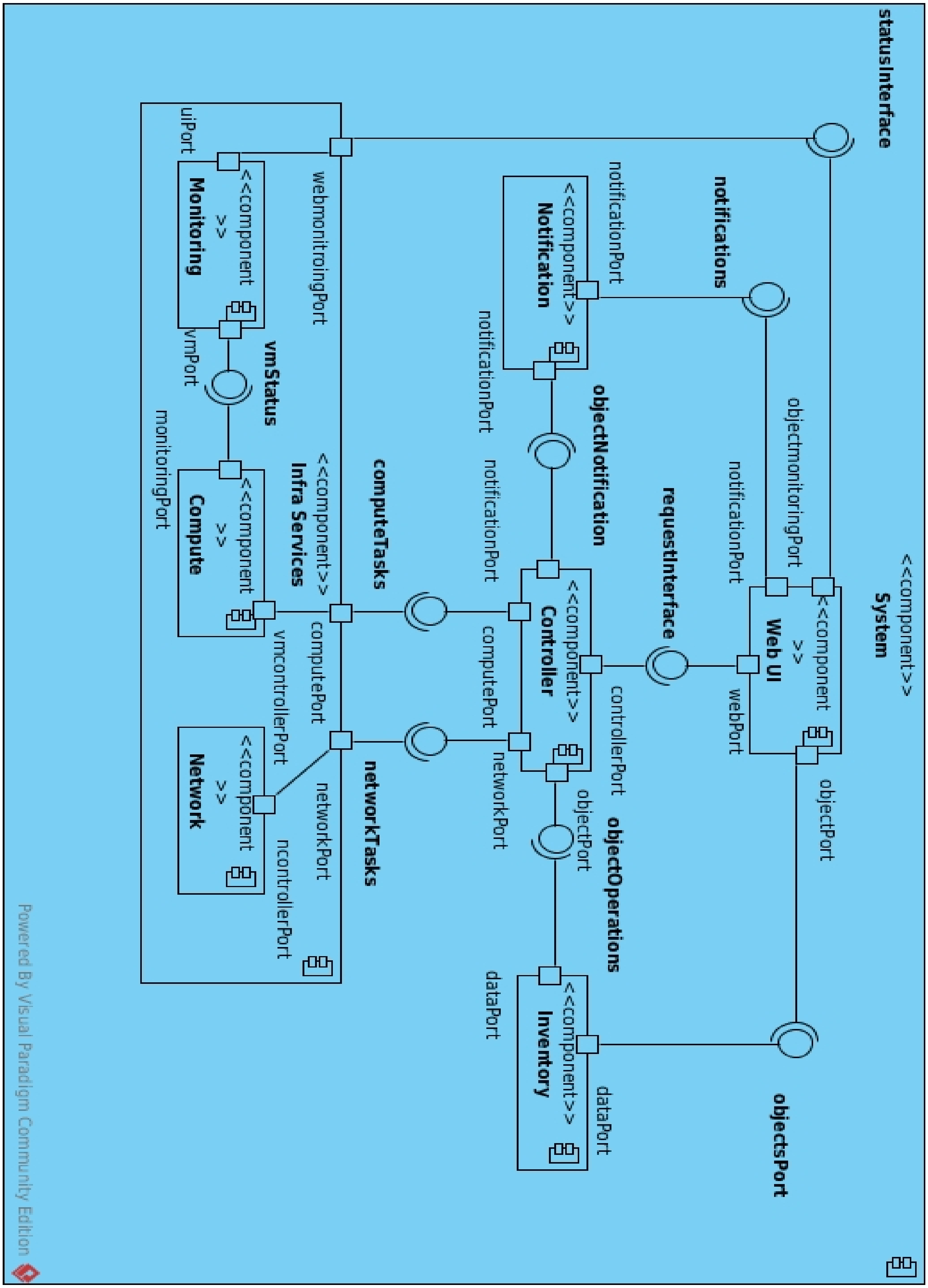
They manage the infrastructure, create and manage virtual resources. They are also used to monitor and collect infrastructure and virtual machines resource usage data.

They are: ***Compute, Network, Monitoring***

3- Infrastructure:

This refers to the physical nodes used as hypervisors to host virtual machines and nodes used to provide shared storage.

(Note: there's also messaging and coordination middle-wares used in Kloudak and will be discussed in chapter 4: Deployment Guide)



<<component>>
System
statusInterface
notifications
<<component>>
Notification
notificationPort
notificationPort
objectmonitoringPort
notificationPort
<<component>>
Web UI
webPort
objectPort
requestInterface
objectNotification
<<component>>
Controller
notificationPort
computePort
networkPort
controllerPort
objectPort
objectOperations
<<component>>
Inventory
dataPort
dataPort
objectsPort
computeTasks
networkTasks
<<component>>
Infra Services
webmonitroingPort
computePort
networkPort
uiPort
<<component>>
Monitoring
vmPort
vmStatus
<<component>>
Compute
monitoringPort
vmcontrollerPort
<<component>>
Network
ncontrollerPort
Powered By Visual Paradigm Community Edition

Basic Concepts:

1- Areas:

An area represents a failover domain and a network subnet. Failover means that all hosts in the same area can be used to restart failed virtual machines due to their hosting hypervisor failure. For that purpose all hosts in an area are connected to all shared storage pools in that area.

2- Storage Pools:

They refer to storage space used to store virtual machine files. No matter what underlying technology you use for shared storage, the only restriction is that each storage pool is mounted in the same path on all hypervisors.

3- JSON-WEB-TOKEN:

or JWT is an authentication & authorization technology where each user have a unique token generated when logged in which contains user information. The user attaches his token in every request to be validated at the server-side.

We use JWT to store user permissions and to distinguish superusers who are allowed to manipulate inventory objects.

3- Authentication & Authorization

End-User Security:

1- Login:

Users can login by posting ‘username’ and ‘password’ to the inventory URL (/login/).

2- Acquiring Token:

Only inventory service contain user information and permissions and for that reason, it generates security token to be used as identification when interacting with other services.

After logging in to inventory, users should obtain their token to be able to interact with other services (controller and notification) from inventory URL (/get_token/) using GET method.

The token should be embedded in controller requests’ header of ‘token’, and as a parameter when connecting to notification. For example (/Workspace-01/<token value>).

3- Token Information:

Information embedded inside user token basically looks like this:

{ ‘username’: <name>,

‘email’: <email>,

<wokspace name>: {

    ‘user_can_add’: <True/False>,

    ‘user_can_edit’: <True/False>,

‘user_can_delete’: <True/False>,

‘vm_can_add’: <True/False>,

‘vm_can_edit’: <True/False>,

‘vm_can_delete’: <True/False>,

‘network_can_add’: <True/False>,

‘network_can_edit’: <True/False>,

‘network_can_delete’: <True/False>

},

…..

}

For each workspace a user belongs to, a permissions dictionary is embedded in the token as a value for a key of the workspace name.

Administrators Security:

1- Superuser Account:

End-user accounts can only view objects in inventory (except for workspaces and users) but cannot add or delete objects directly from inventory as this must only be done by controller. Thus, we build superuser accounts used in internal interactions between services and require no login.

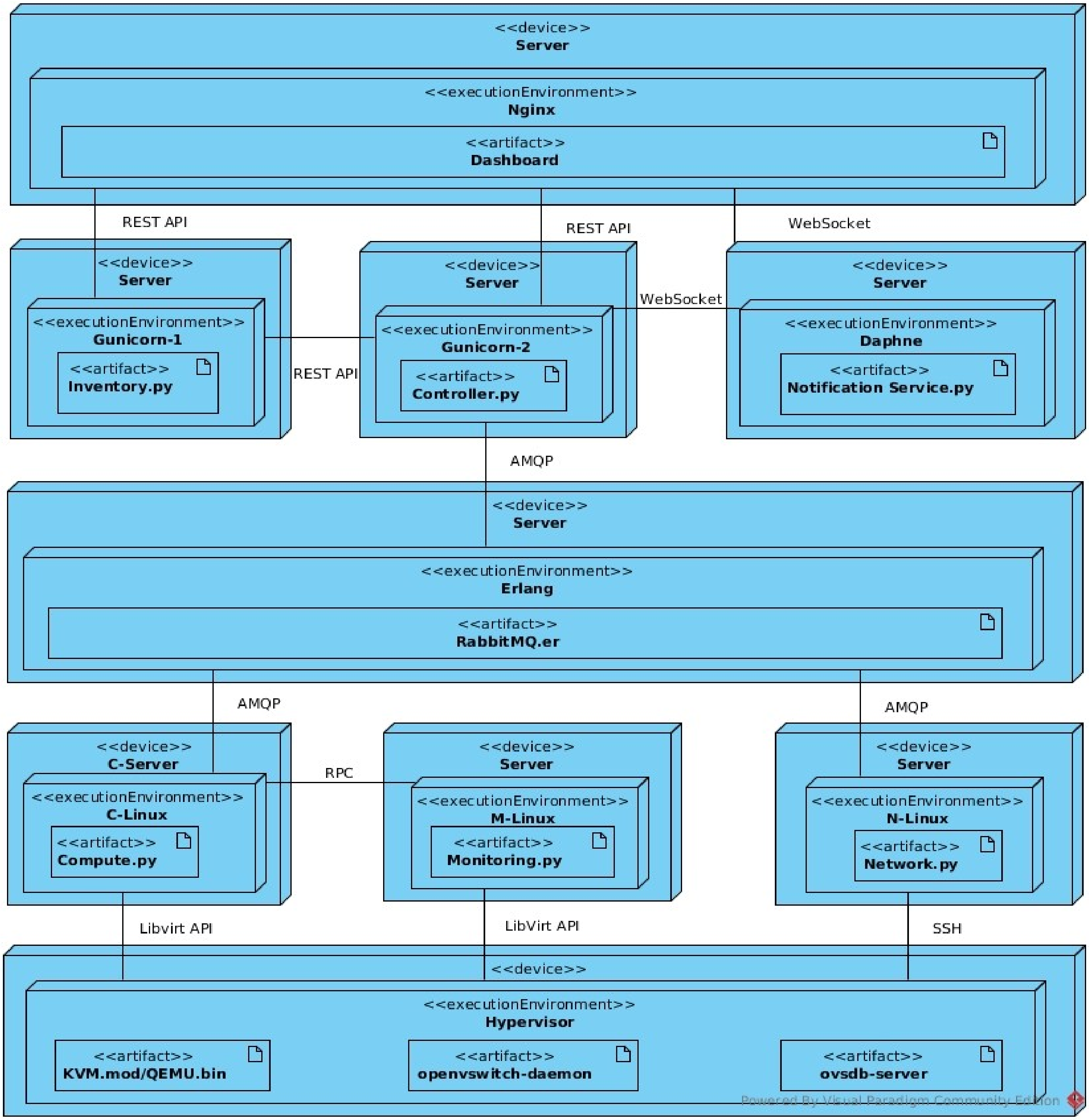
Superuser accounts use a token with a username of the superuser inside each service.

So basically it can look only like this:

{‘username’: <superuser name>}

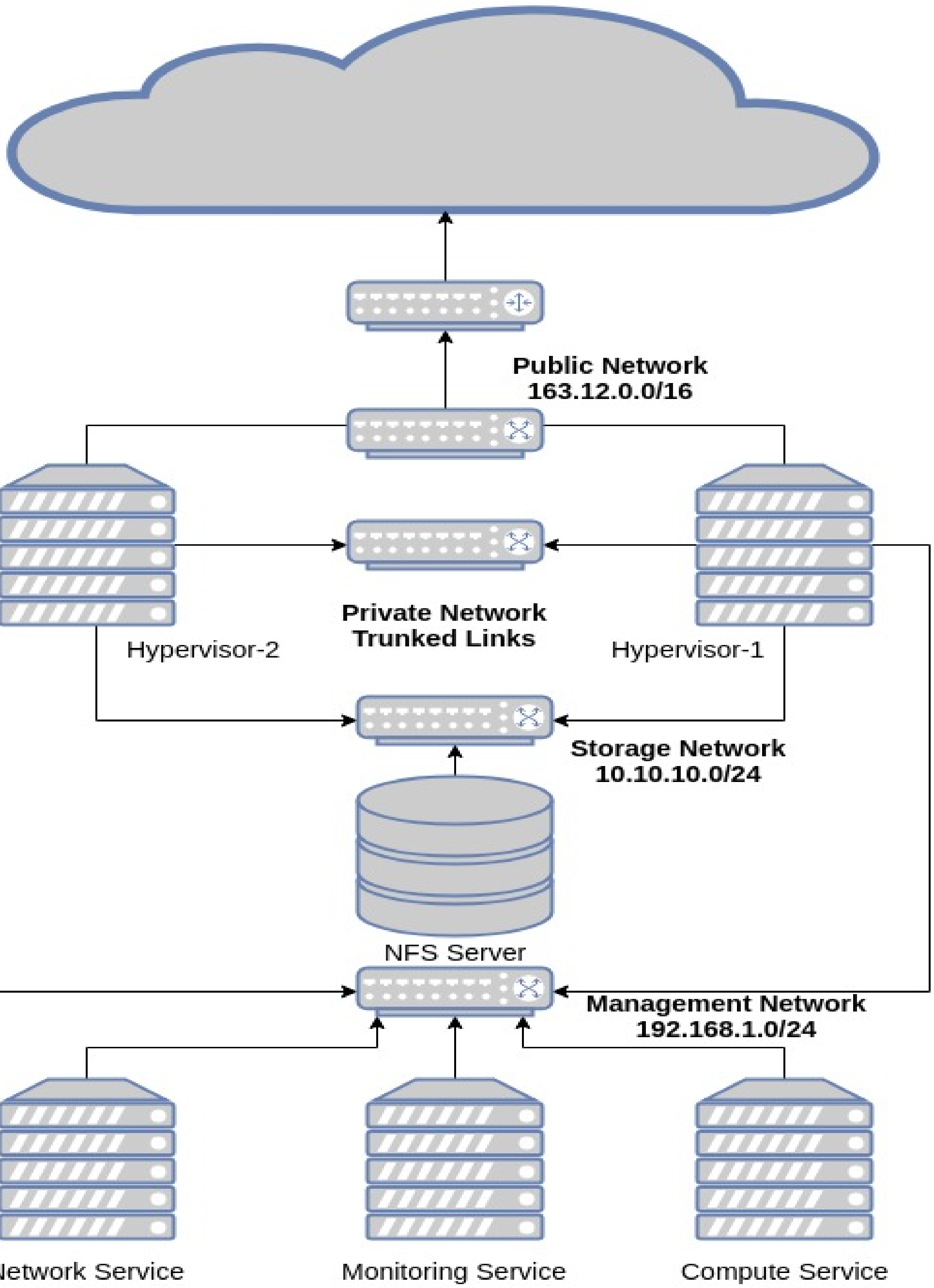
4- Deployment Guide

In this chapter we will walk through a simple deployment of Kloudak.



Infrastructure:

For simplicity the infrastructure will consist of one area with two hypervisors each with four NICs and a single storage pool in it of type NFS. It should look like this:



1- Storage (NFS-Server):

In this walk through we are using Fedora 27 to install the server.

First, install nfs-utils packages:

```
$ sudo dnf install -y nfs-utils
```

We also need to configure the firewall to allow nfs ports:

```
$ sudo firewall-cmd --add-service=nfs --permanent
$ sudo firewall-cmd --reload
```

Now enable and restart the server:

```
$ sudo systemctl enable nfs
$ sudo systemctl restart nfs
```

(Note: sometimes nfs server uses different ports which will need to be allowed manually depending on your case or you can just disable firewall :D).

Second, export directory to NFS:

```
$ cat /etc/exports

/var/lib/nfsroot
10.10.10.0/24(rw,sync,no_root_squash,no_subtree_check)
$ sudo mkdir /var/lib/nfsroot
$ sudo chmod 777 /var/lib/nfsroot
$ sudo exportfs -ra
```

2- Hypervisors (NFS-Server):

We are using KVM as a hypervisor an QEMU for hardware emulation on top of fedora 27.

First, install virtualization group:

```
$ sudo dnf groupinstall virtualization
$ sudo systemctl enable libvirtd
$ sudo systemctl start libvirtd
$ sudo systemctl enable sshd
$ sudo systemctl start sshd
```

3- Network (Open vSwitch):

We will use OVS bridges on hypervisors to connect virtual machines to public and private networks.

Each host should have four NICs:

1- Connected to storage network.

2- Connected to management network.

3- Connected to public network.

4- Connected to private network.

First, install Open vSwitch:

```
$ sudo dnf install openvswitch
$ sudo systemctl enable openvswitch
$ sudo systemctl enable openvswitch
```

Second, configure bridges:

```
$ sudo ovs-vsctl add-br Public-br
$ sudo ovs-vsctl add-br Private-br
$ sudo ovs-vsctl add-port public-br <interface-in-public-network>
```

```
$ sudo ovs-vsctl add-port private-br <interface-in-private-network>
$ sudo ip addr flush dev <interface-in-public-network>
$ sudo ip addr flush dev <interface-in-private-network>
$ sudo ip link set <interface-in-public-network> up
$ sudo ip link set <interface-in-private-network> up
$ sudo ovs-vsctl set port <interface-in-private-network>
vlan_mode=trunk
```

4- Configuring Pools:

First, create the directory where the pool will be mounted:

```
$ sudo mkdir /var/lib/pool-01
```

Second, create an XML file called ‘pool-01.xml’ for the pool:

```
<pool type='netfs'>
	<name>pool-01</name>
	<source>
		<host name='10.10.10.10'/>
		<dir path='/var/lib/nfsroot'/>
	</source>
	<target>
		<path>/var/lib/pool-01</path>
	</target>
</pool>
```

Finally, use virsh to define and start the pool on both hosts:

```
$ sudo virsh pool-define pool-01.xml
$ sudo virsh pool-start pool-01
```

(Execute the previous steps on both hypervisors)

Middle-wares:

We will walk through basic installation of the messaging and coordination middle-wares.

1- RabbitMQ:

RabbitMQ is an enterprise messaging middle-ware. We use it in Kloudak to queue and deliver messages between controller service and automation services. It is also used as RPC middle-ware for the server in monitoring service and client in compute service.

For the installation, we will use Fedora 27 as operating system.

First, install rabbitmq-server:

```
$ sudo dnf install rabbitmq-server
```

Second, enable and start the server:

```
$ sudo systemctl enable rabbitmq
$ sudo systemctl start rabbitmq
```

2- Redis:

Redis is used as a back-end for the notification service to provide highly scalable real-time web application.

First, install redis:

```
$ sudo dnf install redis
```

Second, enable and start the server:

```
$ sudo systemctl enable redis
$ sudo systemctl start redis
```

3- Zookeeper:

Zookeeper provides many requirements for distributed systems such as: leader election and distributed coordination premitives. We use it in the monitoring service to elect a leader for the monitoring cluster which would be responsible for assigning tasks to the rest of the cluster using Celery.

First, install zookeeper:

```
$ sudo dnf install zookeeper
```

Second, create the server configuration file and name it zoo1.cfg:

tickTime=2000

initLimit=5

syncLimit=2

dataDir=/var/lib/zookeeper/zoo1

clientPort=2181

server.1=localhost:2666:3666

Finally, enable and start the server:

```
$ sudo systemctl enable zookeeper
$ sudo systemctl start zookeeper
$ zkServer start zoo1.cfg
```

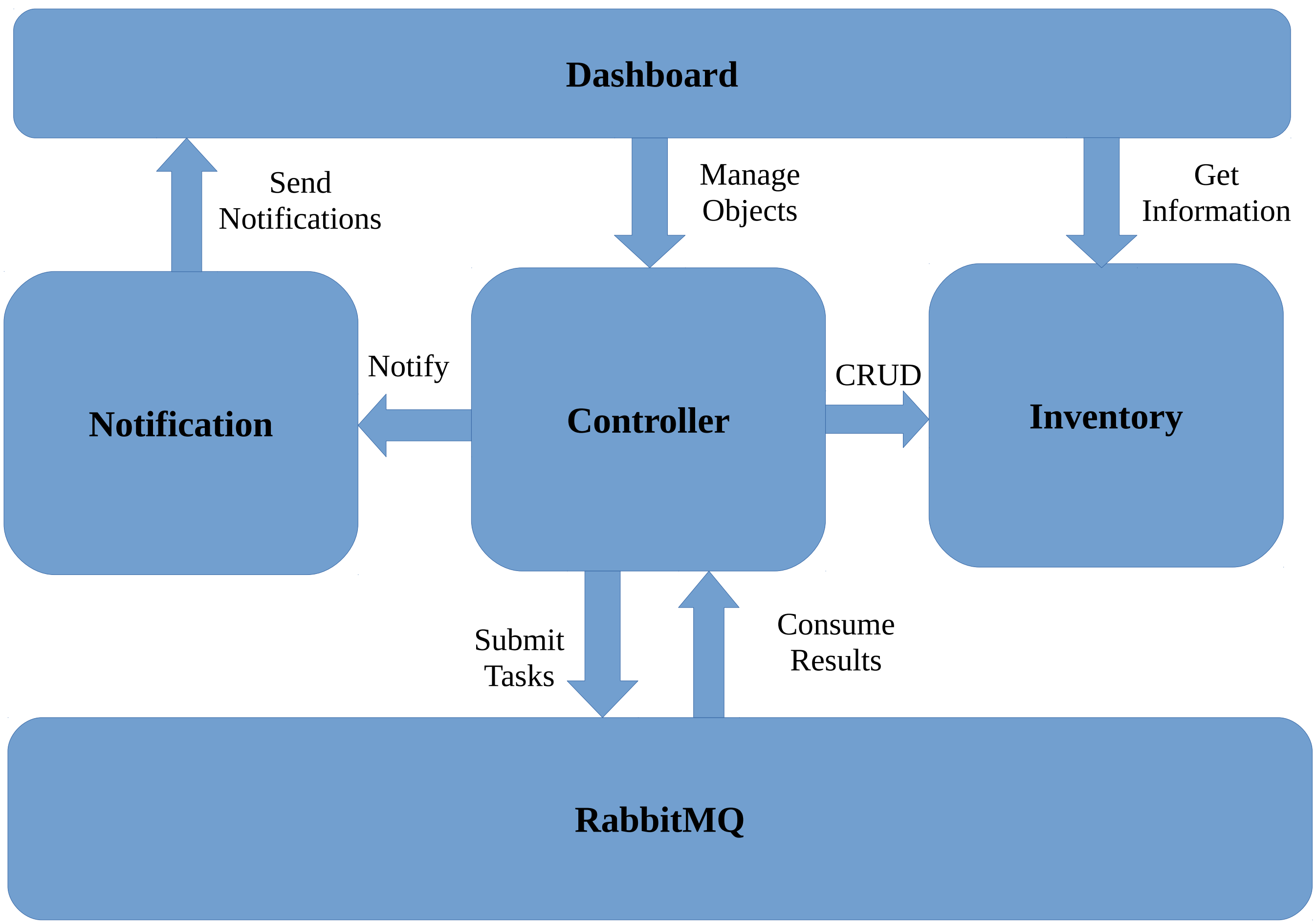
Backend Services:

Back-end services are written in Python 3.6 using Django 2.0. Each service is using PostgreSQL as a database.

To obtain Kloudak source code from github:

$ git clone git://github.com/m-motawea/Kloudak.git

The following diagram shows the relation between these services:



1- Database (PostgreSQL):

Here, we will install one database server, but you should install a separate database server for each service.

First, install these packages:

```
$ sudo dnf install postgresql postgresql-server postgresql-devel
```

Second, enable and start the server:

```
$ sudo systemctl enable postgres
$ sudo systemctl start postgres
```

2- Inventory:

The next commands are executed inside Inventory directory in Kloudak source code except for database creation commands are executed on PostgreSQL server.

First, install the dependencies:

$ sudo pip3.6 install -r Inventory_Service/requirements.txt

Second, install Gunicorn server to run the application:

$ sudo pip3.6 install gunicorn

Third, edit the settings.py file inside Inventory_Service/Inventory_Service in database section use the following:

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql_psycopg2',

```
        'NAME': 'inventory',

        'USER': 'inv_user',

        'PASSWORD': 'Maglab123!',

        'HOST': '172.17.0.1',

        'PORT': '5432',

    }

        }
```

Fourth, configure the inventory database inside PostgreSQL:

```
$ sudo su postgres
$ psql
$ CREATE DATABASE inventory;
$ CREATE USER inv_user WITH PASSWORD ‘Maglab123!’;
$ ALTER ROLE inv_user SET client_encoding TO 'utf8';
$ ALTER ROLE inv_user SET default_transaction_isolation TO 'read committed';
$ GRANT ALL ON DATABASE inventory TO inv_user;
```

Fifth, create the database schema using django migrations:

```
$ python3.6 Inventory_Service/manage.py makemigrations
$ python3.6 Inventory_Service/manage.py migrate
```

Sixth, create a superuser:

```
$ python3.6 Inventory_Service/manage.py createsuperuser
Username: maged
Email: <whatever>
Password: <whatever>
```

Finally, start the server:

$ Inventory_Service/start.sh

3- Controller:

The next commands are executed inside Contoller directory in Kloudak source code except for database creation commands are executed on PostgreSQL server.

First, install the dependencies:

$ sudo pip3.6 install -r Controller_Service/requirements.txt

Second, install Gunicorn server to run the application:

$ sudo pip3.6 install gunicorn

Third, edit conf.json inside Controller_Service/ControllerAPI with the following:

```
{
    "inventory": "http://172.17.0.1:5000/",
    "notification": "localhost",
    "broker": "172.17.0.1",
    "retries": 2,
    "wait": 1
}
```

Fourth, edit the databse section inside settings.py file inside Controller_Service -/Contoller_Service with the following:

```
DATABASES = {

        'default': {

        'ENGINE': 'django.db.backends.postgresql_psycopg2',

        'NAME': 'controller',

        'USER': 'cont_user',

        'PASSWORD': 'Maglab123!',

        'HOST': '172.17.0.1',

        'PORT': '5432',

    }}
```

Fifth, configure the controller database inside PostgreSQL:

```
$ sudo su postgres
$ psql
$ CREATE DATABASE controller;
$ CREATE USER cont_user WITH PASSWORD ‘Maglab123!’;
$ ALTER ROLE cont_user SET client_encoding TO 'utf8';
$ ALTER ROLE cont_user SET default_transaction_isolation TO 'read committed';
$ GRANT ALL ON DATABASE controller TO cont_user;
```

Sixth, create the database schema using django migrations:

```
$ python3.6 Controller_Service/manage.py makemigrations
$ python3.6 Controller_Service/manage.py migrate
```

Seventh, create a superuser:

```
$ python3.6 Controller_Service/manage.py createsuperuser
Username: maged
Email: <whatever>
Password: <whatever>
```

Eighth, start notification consumers:

```
$ python3.6 Controller_Service/QueueMonitoring vm_notification_consumer.py
$ python3.6 Controller_Service/QueueMonitoring network_notification_consumer.py
```

Finally, start the server:

```
$ Controller_Service/start.sh
```

4- Notification:

The next commands are executed inside Notification directory in Kloudak source code.

First, install the dependencies:

```
$ sudo pip3.6 install -r mysite/requirements.txt
```

Second, install Daphne server to run the application:

```
$ sudo pip3.6 install daphne
```

Third, create the database schema using django migrations:

```
$ python3.6 mysite/manage.py makemigrations
$ python3.6 mysite/manage.py migrate
```

Fourth, create a superuser:

```
$ python3.6 mysite/manage.py createsuperuser
Username: maged
Email: <whatever>
Password: <whatever>
```

Fifth, edit channels layer section in settings.py in mysite/mysite with the following:

```
CHANNEL_LAYERS = {

    'default': {

        'BACKEND': 'channels_redis.core.RedisChannelLayer',

        'CONFIG': {

            "hosts": [('172.17.0.1', 6379)],

        },

    },

}
```

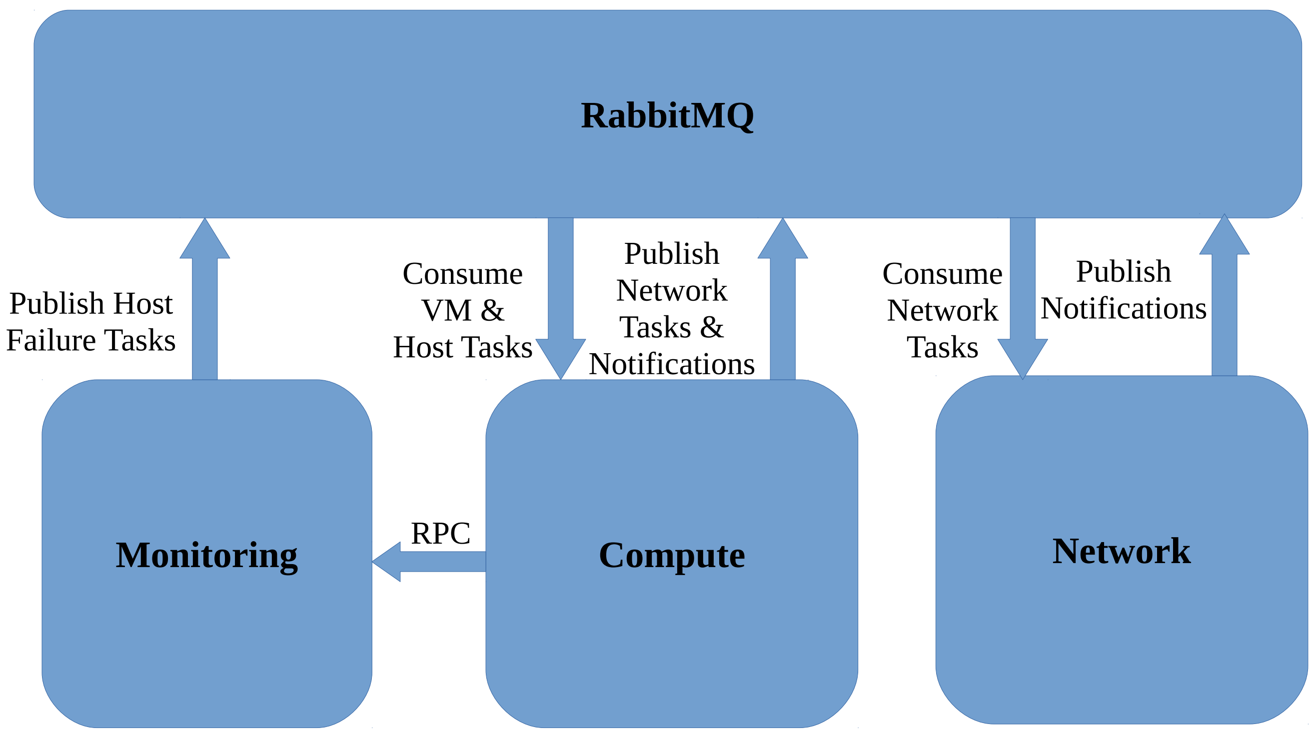
Finally, start the server:

```
$ mysite/start.sh
```

Automation Services:

Back-end services are written in Python 3.6 and they are the layer that interacts with the infrastructure to create and manage Virtual Machines and networks lifecycle as well as monitoring the infrastructure.

The following diagram describes the relation between them:



1- Monitoring:

The next commands are executed inside Monitoring directory in Kloudak source code except for database creation commands are executed on PostgreSQL server.

First, install the following packages:

```
$ sudo dnf install libvirt libvirt-devel
```

Second, install the dependencies:

```
$ sudo pip3.6 install -r requirements.txt
```

Third, configure the database:

```
$ sudo su postgres
$ psql
$ CREATE DATABASE monitor;
$ CREATE USER mon_admin WITH PASSWORD ‘Maglab123!’;
$ GRANT ALL ON DATABASE monitor TO mon_admin;
```

Fourth, edit conf.json file with the following:

```
{
    "broker": "172.17.0.1",
    "database": "172.17.0.1",
        "time":10
}
```

Fifth, create the database schema:

$ python3.6 db_setup.py

Sixth, edit pop_db.py file with the following:

```
#!/usr/bin/python3.6
from orm_schema import Host, Area, Pool
from orm_io import dbIO
io = dbIO('localhost')
a = Area(area_name='Area-01')
io.add([a])

a = io.query(Area, area_name='Area-01')[0]
h1 = Host(
host_name='kvm-1',
host_ip='192.168.1.7',
host_cpus=2,
host_memory=8,
host_free_memory=8,
state=True,
area_id=a.area_id)
h2 = Host(
host_name='kvm-2',
host_ip='192.168.1.8',
host_cpus=2,
host_memory=8,
host_free_memory=8,
state=True,
area_id=a.area_id
)
io.add([h1, h2])

p1 = Pool(
pool_name='pool-01',
pool_path='/var/lib/pool-01/',
pool_size=20,
pool_free_size=20,
area_id=a.area_id
)
io.add([p1])
```

Seventh, populate the database with the previous information:

```
$ python3.6 pop_db.py
```

Eighth, install and start a zookeeper client

```
$ sudo dnf install zookeeper
$ sudo systemctl start zookeeper
$ sudo zkCli.sh -server localhost:2181
```

Ninth, start a celery worker:

```
$ celery -A tasks worker --loglevel=info –hostname=h1
```

Tenth, start the RPC server:

```
$ python3.6 rpcServer.py
```

Eleventh, add monitoring ssh key to trusted keys on both hypervisors:

```
$ ssh-copy-id root@kvm-1
$ ssh-copy-id root@kvm-2
```

Finally, start the monitor service:

```
$ python3.6 monitor.py
```

2- Compute:

The next commands are executed inside Compute directory in Kloudak source code except for database creation commands are executed on PostgreSQL server.

First, install the following packages:

```
$ sudo dnf install libvirt libvirt-devel
```

Second, install the dependencies:

$ sudo pip3.6 install -r Worker/requirements.txt

Third, configure the database:

```
$ sudo su postgres
$ psql
$ CREATE DATABASE compute;
$ CREATE USER comp_admin WITH PASSWORD 'Maglab123!';
$ GRANT ALL ON DATABASE compute TO comp_admin;
```

Fourth, edit Worker/conf.json file with the following:

```
{

    "broker": "172.17.0.1",

    "database": "172.17.0.1",

}
```

Fifth, create the database schema:

$ python3.6 Worker/db_setup.py

Sixth, edit Worker/pop_db.py with the following:

```
#!/usr/bin/python3.6

from lib2.orm_schema import Host, Area, Pool, Template
from lib2.base import dbIO
io = dbIO('localhost')
a = Area(
area_name='Area-01',
```

```
area_gw='163.12.0.1'
)
io.add([a])
t = Template(
template_name='Template-01',
template_path='/var/lib/pool-01/',
template_ifname='eth0'
)
a = io.query(Area, area_name='Area-01')[0]
h1 = Host(
host_name='kvm-1',
host_ip='192.168.1.7',
host_cpus=2,
host_memory=8,
host_free_memory=8,
state=True,
area_id=a.area_id
)
h2 = Host(
host_name='kvm-2',
host_ip='192.168.1.8',
host_cpus=2,
host_memory=8,
host_free_memory=8,
state=True,
area_id=a.area_id
)
io.add([h1, h2])
p1 = Pool(
pool_name='pool-01',
pool_path='/var/lib/pool-01/',
pool_size=20,
pool_free_size=20,
area_id=a.area_id
)
io.add([p1])
```

Fifth, create the database schema:

```
$ python3.6 Worker/pop_db.py
```

Sixth, download and extract Fedora cloud image to the pool and name it Template-01.raw from the following url:

https://download.fedoraproject.org/pub/fedora/linux/releases/28/Cloud/x86_64/images/Fedora-Cloud-Base-28-1.1.x86_64.raw.xz

Seventh, add the ssh key to trusted keys on both hypervisors:

$ ssh-copy-id root@kvm-1
$ ssh-copy-id root@kvm-2

Eighth, run the rollbackWorker:

$ python3.6 Worker/rollbackWorker.py

Finally, start the main worker:

$ python3.6 Worker/worker.py

3- Network:

The next commands are executed inside Network directory in Kloudak source code except for database creation commands are executed on PostgreSQL server.

First, install the dependencies:

$ sudo pip3.6 install -r VLAN/requirements.txt

Second, configure the database:

$ sudo su postgres
$ psql
$ CREATE DATABASE network;
$ CREATE USER net_admin WITH PASSWORD ‘Maglab123!’;
$ GRANT ALL ON DATABASE network TO net_admin;

Third, edit Worker/conf.json file with the following:

```
{

    "broker": "172.17.0.1",

    "database": "172.17.0.1",

}
```

Fourth, create the database schema:

```
$ python3.6 VLAN/db_setup.py
```

Fifth, edit VLAN/pop_db.py with the following:

```python
#/usr/bin/python3.6

from lib.orm_schema import base, Area, Host
from lib.orm_io import dbIO
from config import get_config
conf_dict = get_config('conf.json')
io = dbIO(conf_dict['database'])
a = io.query(Area, area_name='Area-01')[0]
h1 = Host(
host_name='kvm-1',
host_ip='192.168.1.7',
state=True,
area_id=a.area_id
)
h2 = Host(
host_name='kvm-2',
host_ip='192.168.1.8',
state=True,
area_id=a.area_id
)
io.add([h1, h2])
```

Sixth, populate the database:

```
$ python3.6 VLAN/pop_db.py
```

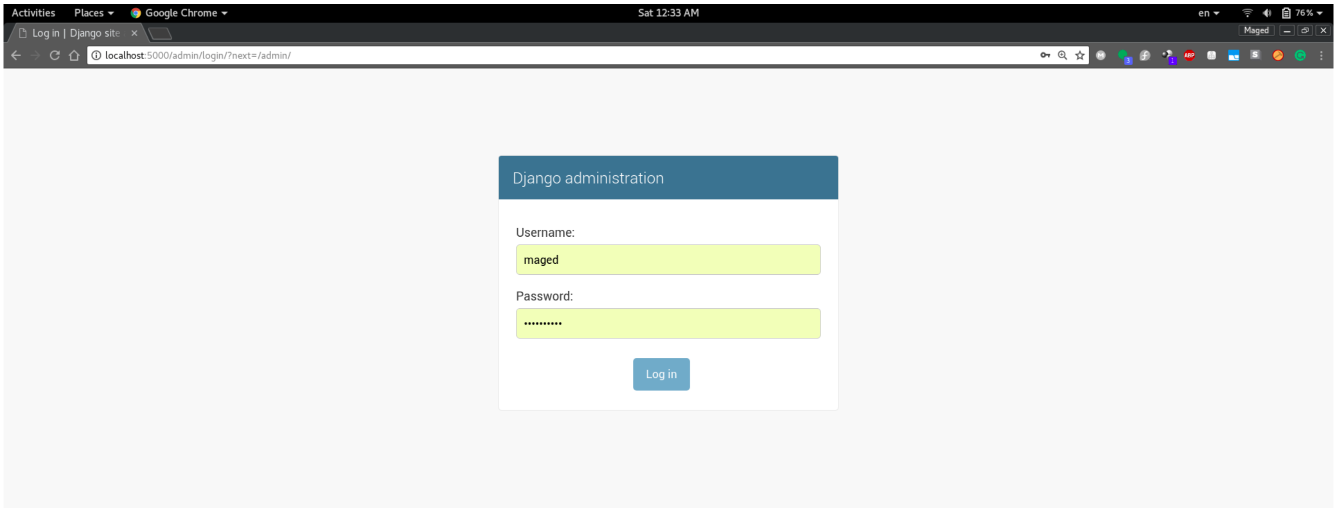
Seventh, run the rollback worker:

```
$ python3.6 VLAN/rollbackWorker.py
```

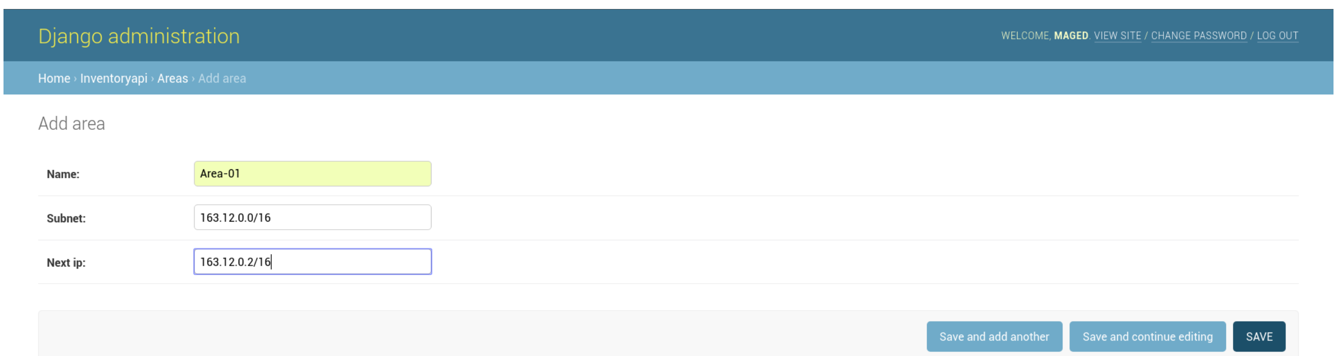
Finally, start the main service worker:

```
$ python3.6 VLAN/worker.py
```

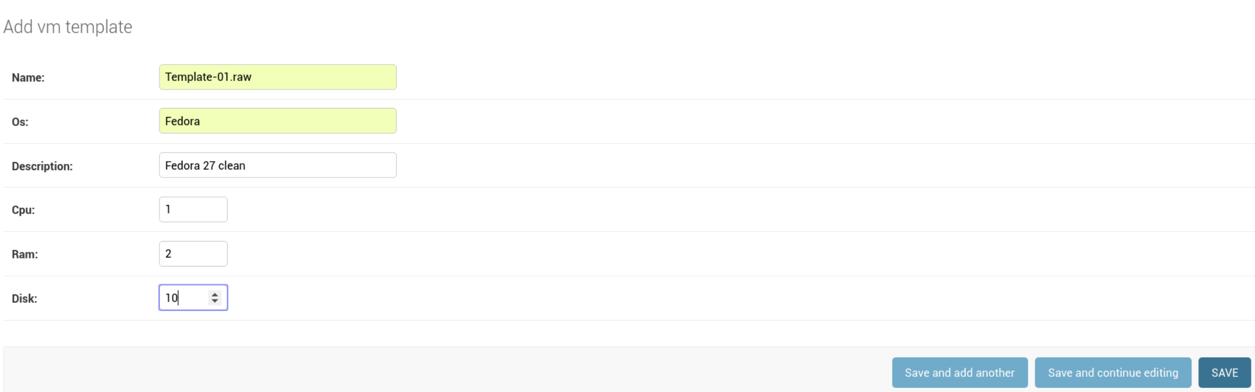
Now, open a web browser and go to inventory administration page and login with superuser credentials.



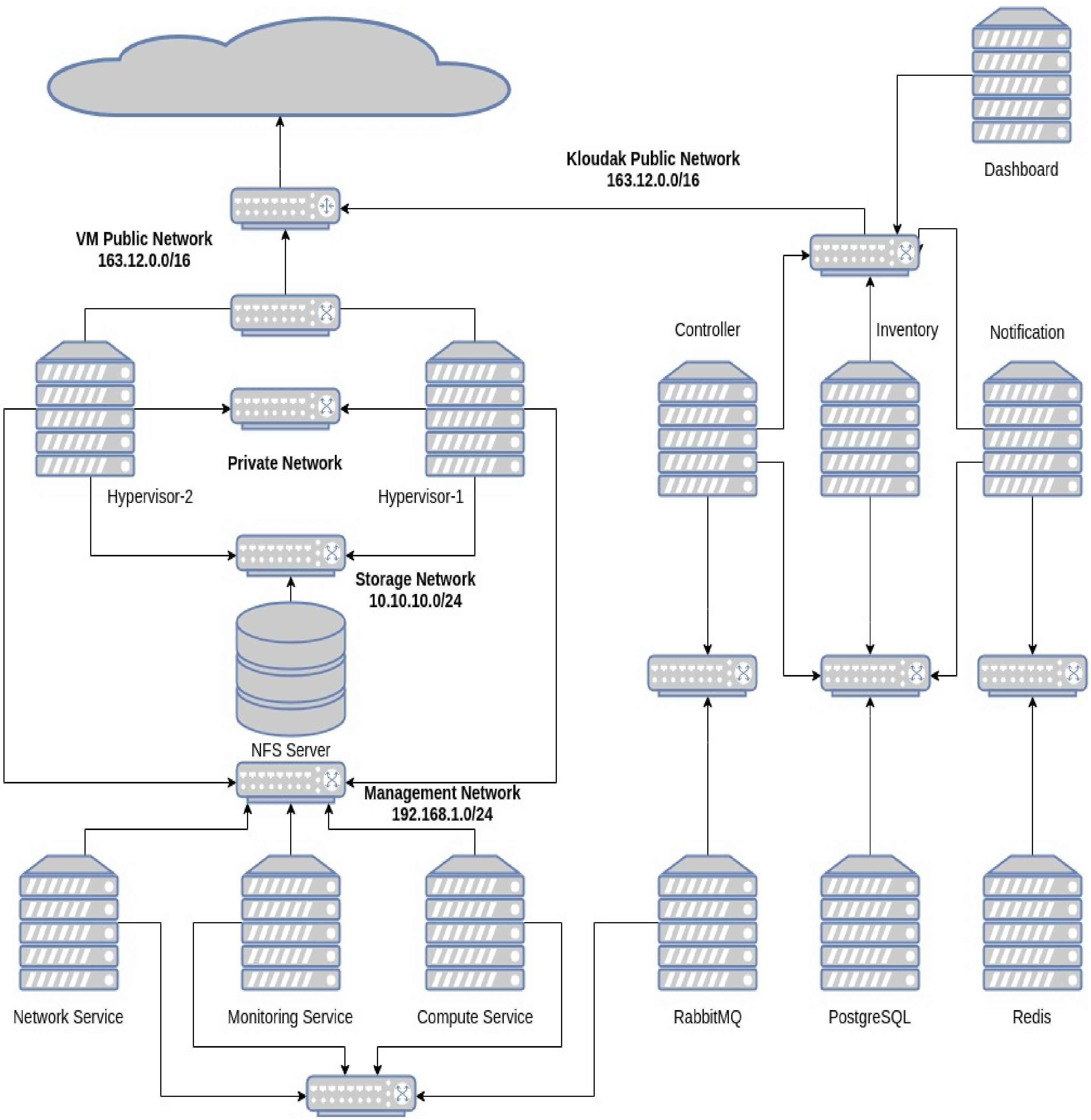
Choose add next to Areas to create a new one with following data:



Choose add next to Templates to create a new one with the following data:



Your deployment should look like this:



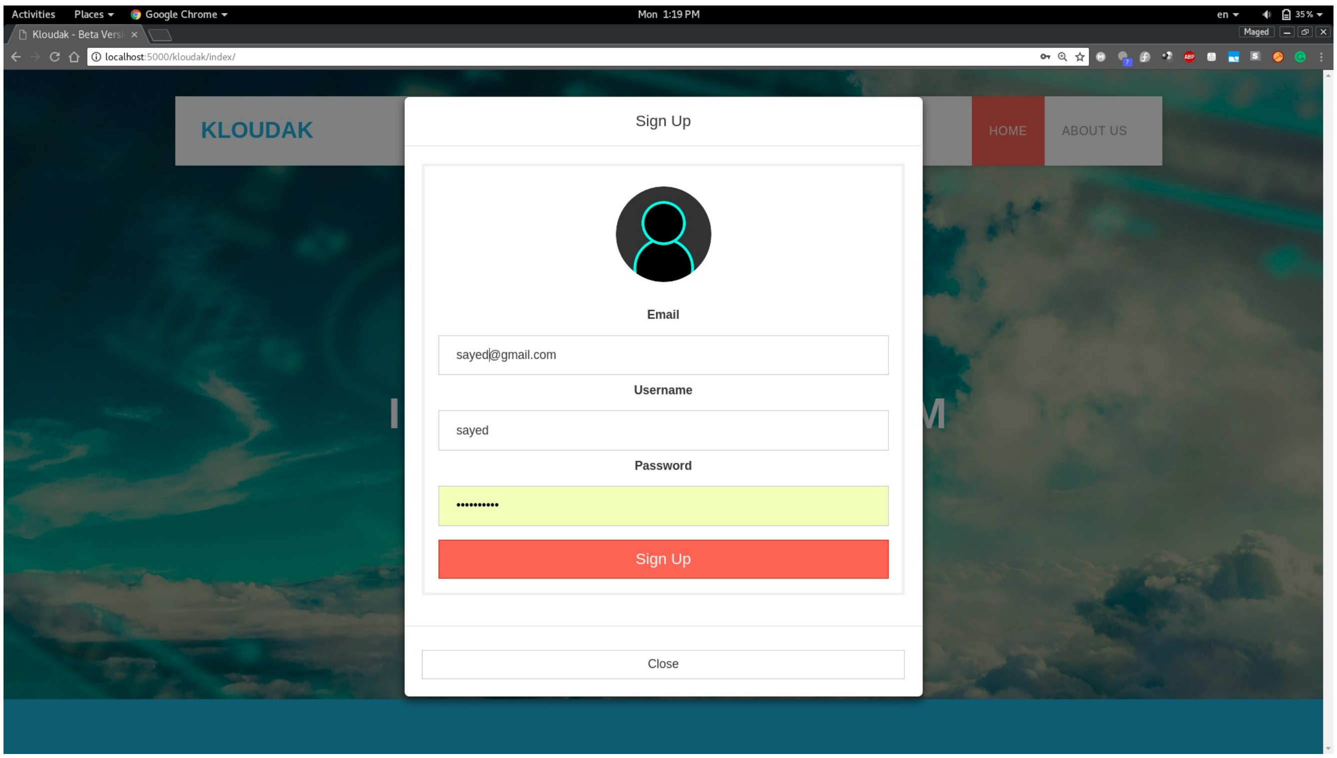
5- End-User Guide

Using the Dashboard:

The dashboard is a web application in the inventory service.

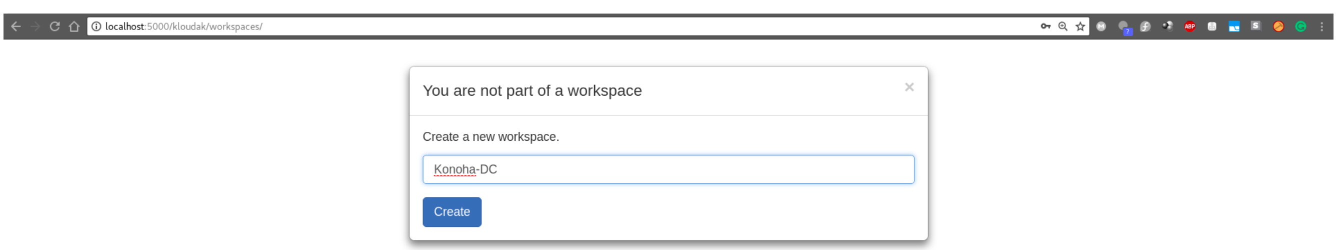
1- User Login/Sign up:

Index URL: ('/kloudak/index/')

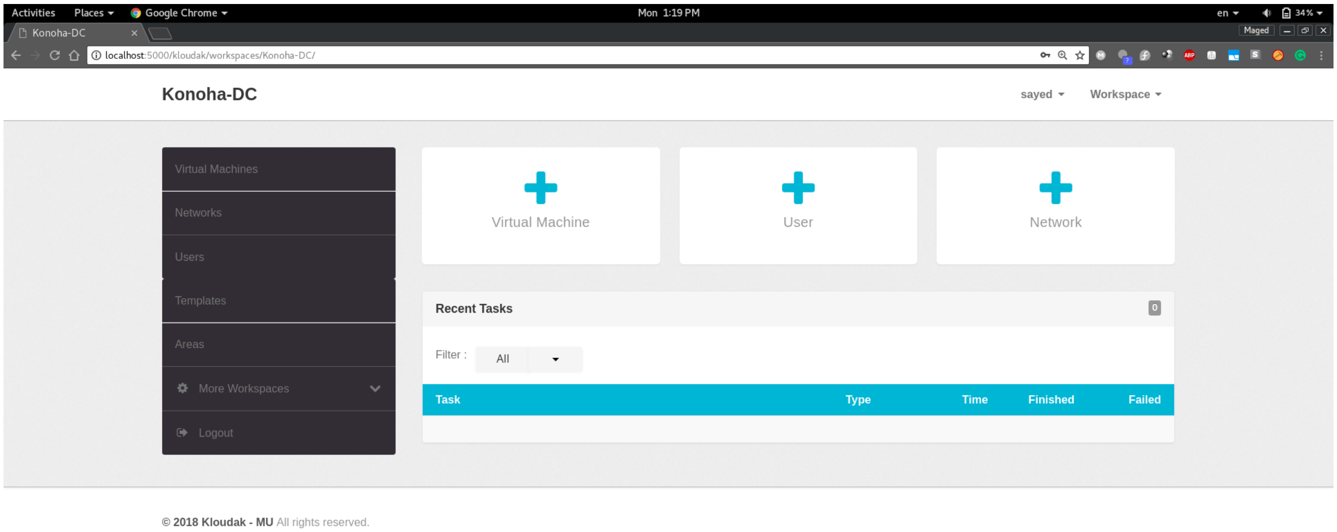


2- Creating your First Workspace:

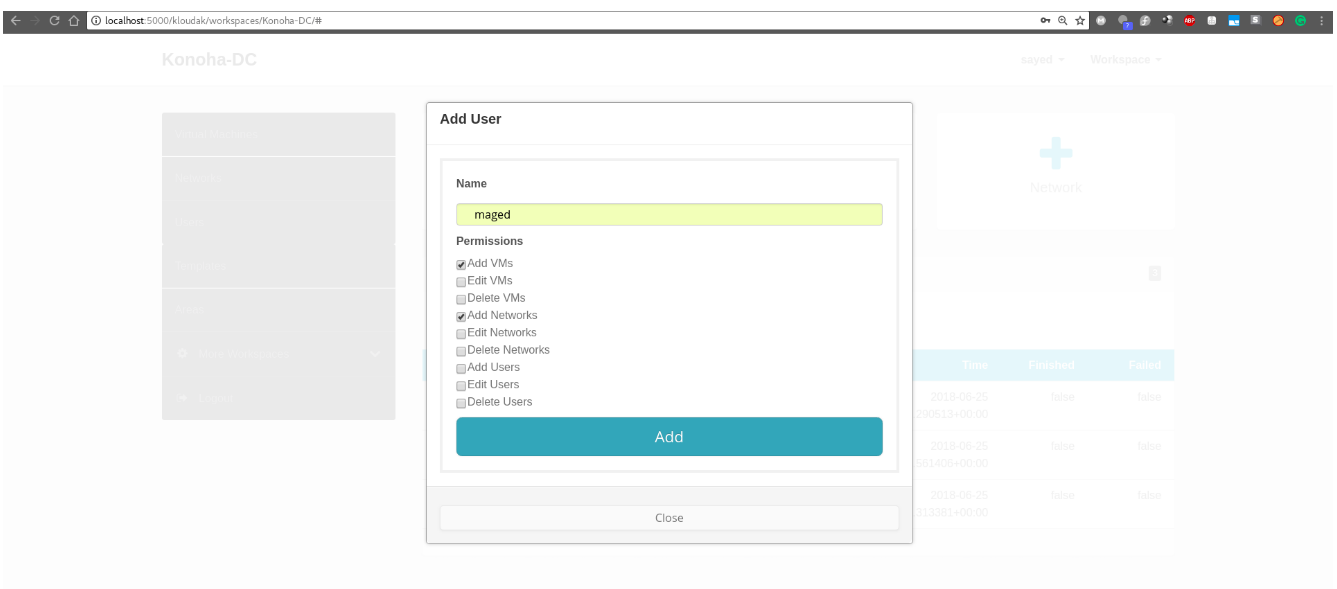
After sign up, you will be directed to create your first workspace.



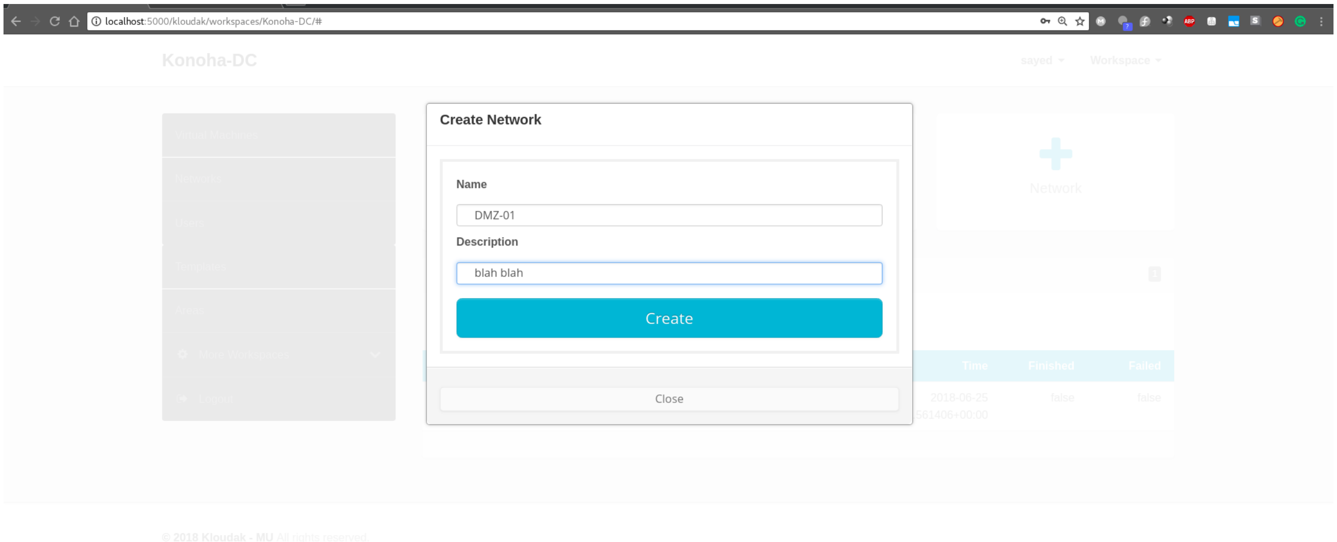
you will be redirected to the workspace dahsboard where you can create and delete objects



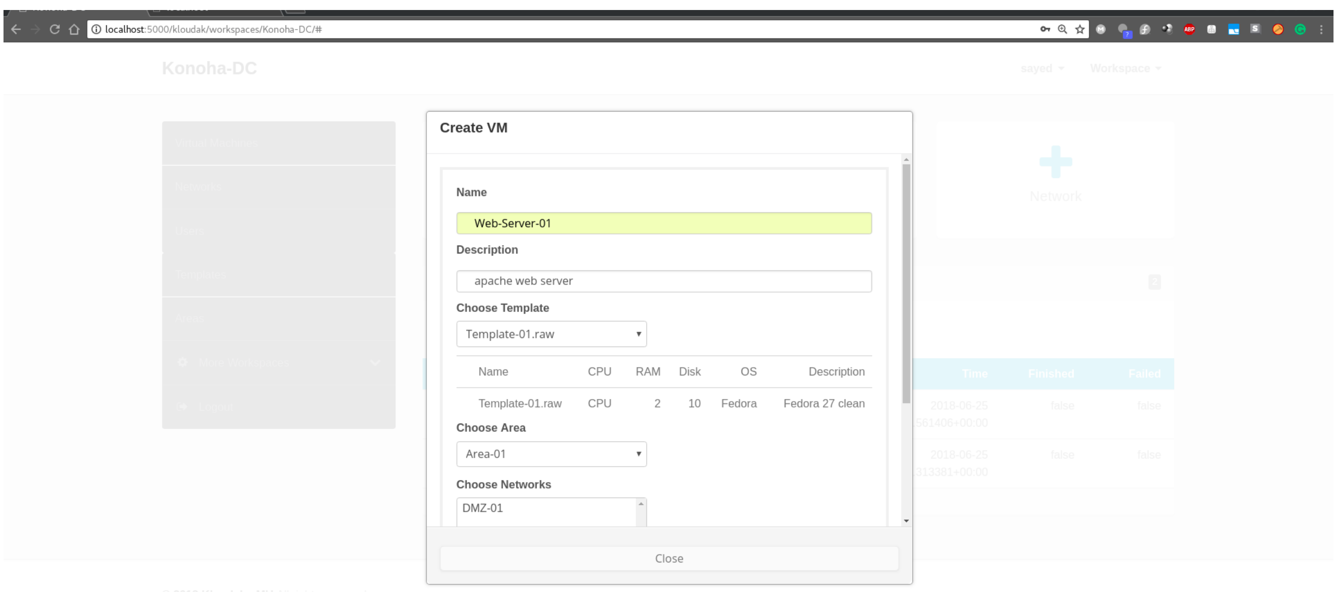
3- Adding Users:

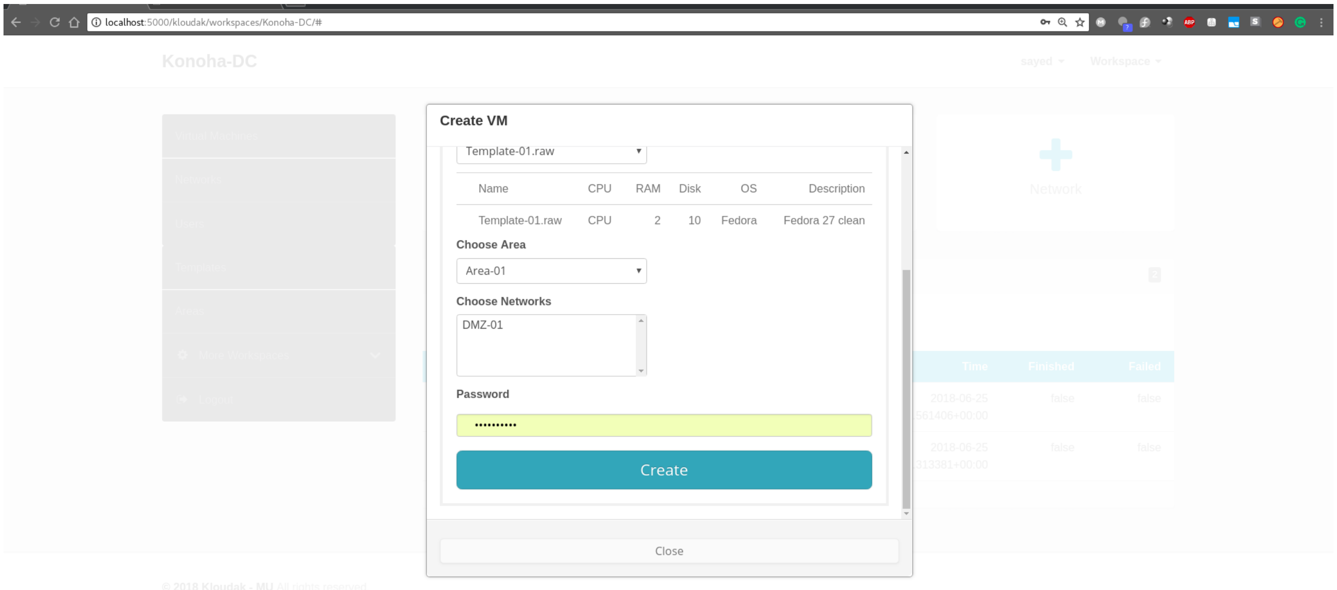


4- Creating Networks:

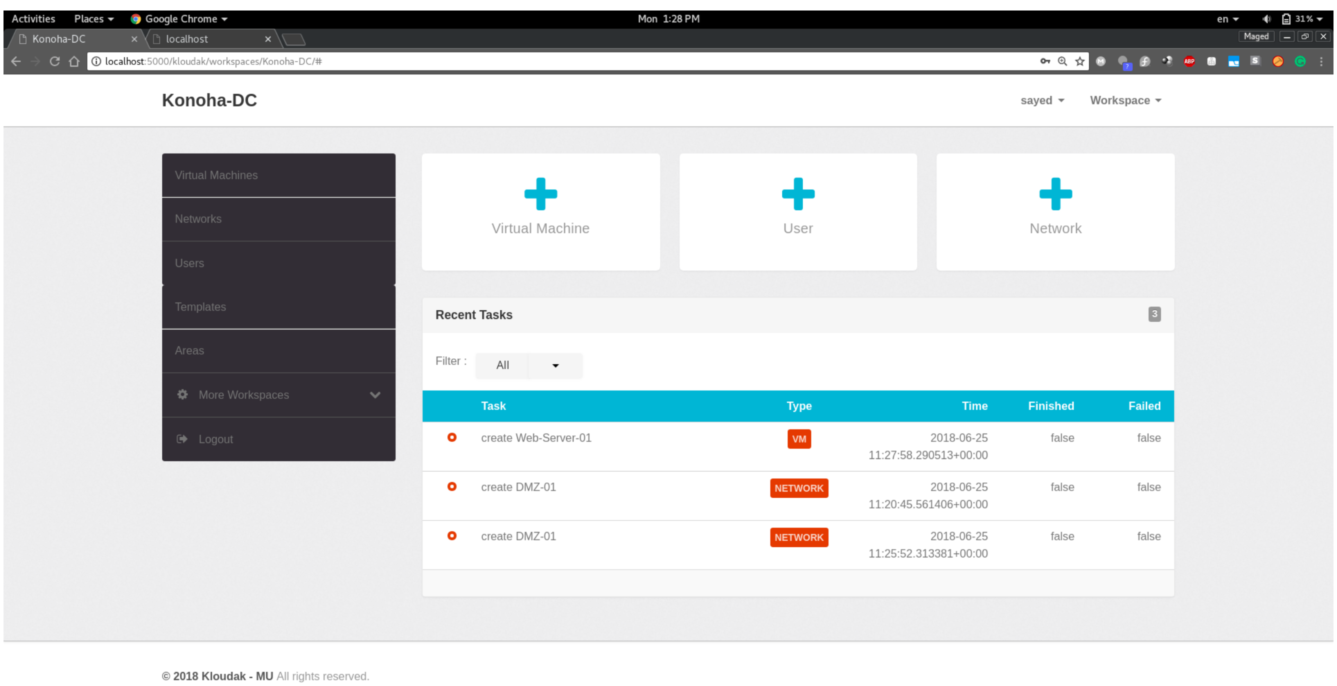


5- Creating Virtual Machines:





(note: 'Recent Tasks' in the dashboard)



6- Inventory

Roles:

1- User Login/Sign up:

Inventory service is considered the identity provider of Kloudak platform. It handles all user related operation.

For login users use POST to send username and password with the same ids to the inventory URL (/login/).

For sign up use POST to send username, password and email with the same ids to the inventory URL (/signup/).

2- Generating Tokens:

Once a user is logged in, he is redirected to a workspace which he is part of or, to create a new workspace if he is not part of any workspaces.

To start interact with the objects in a workspace, a user needs an authentication token to use it as part of his requests to controller and notification services.

To obtain the token use GET to the inventory URL (/get_token/).

3- Data Storage:

Inventory is mainly a data storage as the name suggests. It stores all information used by End-users which is:
- Templates:
Templates describe the SW & HW which would be the same in its children VMs.
    name: (template name)
    OS: (Operating system)
    description: (small description of the template)
    CPU: (number of CPUs)

RAM: (memory size in gigabytes)
disk: (hard disk size in gigabytes)

- Areas:
    name: (area name)
    subnet: (area subnet)
    next available IP
- Workspaces:

A collection of users and resources which functions in a similar way to an organization.

    name: (workspace name)

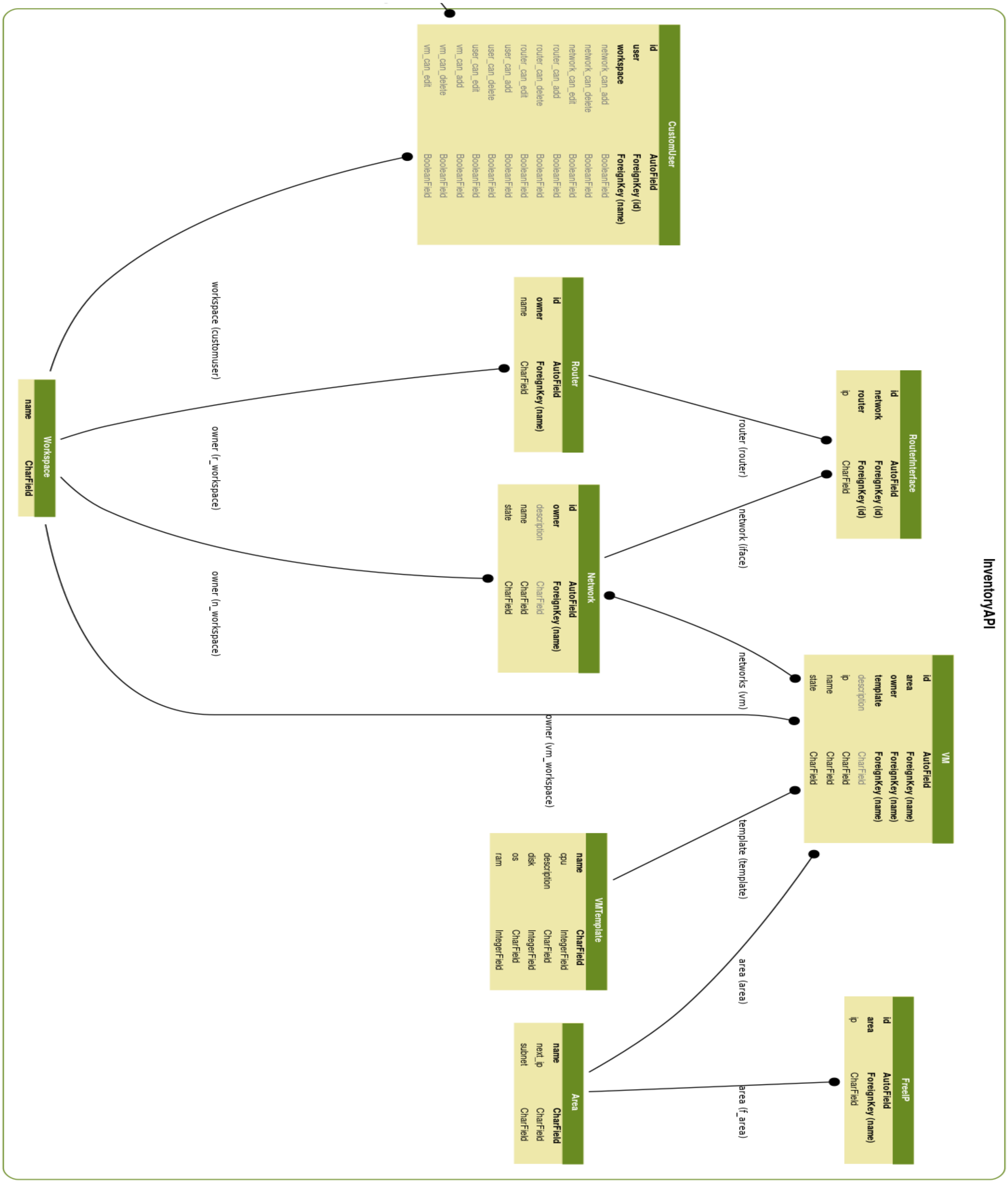
- CustomUsers:

It is basically a user profile in a workspace which contains his permissions

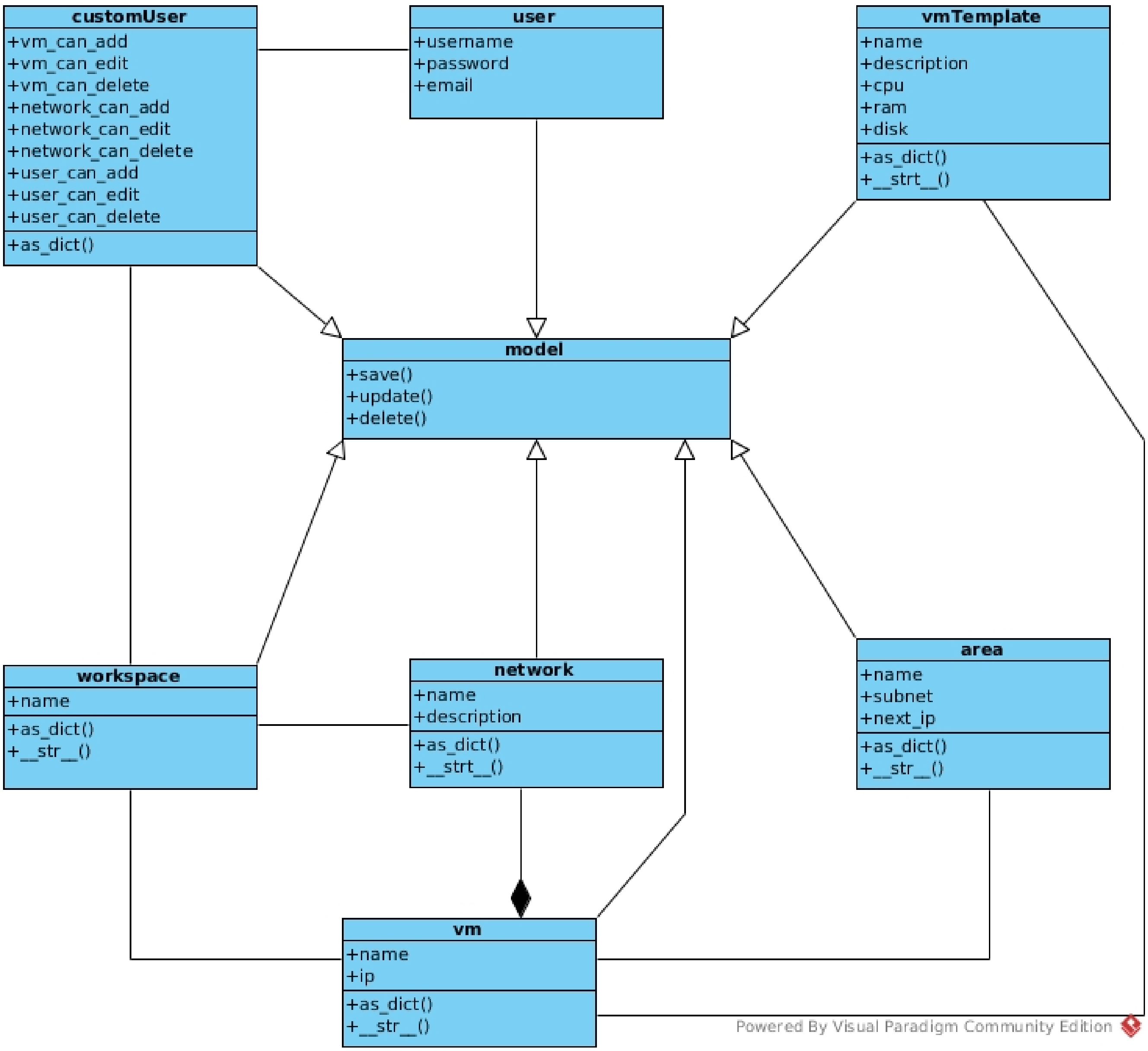
    user: (a foreign key to the user account associated with this profile)
    workspace: (a foreign key to the parent workspace)
    vm_can_add: (boolean which describes the permission)
    vm_can_edit: (boolean which describes the permission)
    vm_can_delete: (boolean which describes the permission)
    … the same goes for network and user permissions

- Networks:
    name: (network name)
    owner: (a foreign key to the parent workspace)
    description: (small description for the network)
- VMs:
    name: (vm name)
    owner: (a foreign key to the parent workspace)
    description: (small description for the network)
    ip: (vm ip address)
    area: (a foreign key to the area which the vm is part of)
    template: (a foreign key to the base template for the vm)
    networks: (many-to-many field refers to networks connected to the vm)

ERD:


InventoryAPI
CustomUser
id AutoField
user ForeignKey (id)
workspace ForeignKey (name)
network_can_add BooleanField
network_can_delete BooleanField
network_can_edit BooleanField
router_can_add BooleanField
router_can_delete BooleanField
router_can_edit BooleanField
user_can_add BooleanField
user_can_delete BooleanField
user_can_edit BooleanField
vm_can_add BooleanField
vm_can_delete BooleanField
vm_can_edit BooleanField
Router
id AutoField
owner ForeignKey (name)
name CharField
RouterInterface
id AutoField
network ForeignKey (id)
router ForeignKey (id)
ip CharField
Workspace
name CharField
Network
id AutoField
owner ForeignKey (name)
description CharField
name CharField
state CharField
VM
id AutoField
area ForeignKey (name)
owner ForeignKey (name)
template ForeignKey (name)
description CharField
ip CharField
name CharField
state CharField
VMTemplate
name CharField
cpu IntegerField
description CharField
disk IntegerField
os CharField
ram IntegerField
Area
name CharField
next_ip CharField
subnet CharField
FreeIP
id AutoField
area ForeignKey (name)
ip CharField
workspace (customuser)
owner (r_workspace)
owner (n_workspace)
router (router)
network (iface)
networks (vm)
owner (vm_workspace)
template (template)
area (area)
area (f_area)

Class Diagram:



7- Controller

Roles:

1- Object Manipulation:

Users can only view information stored in inventory (except for workspaces and users).

To create, modify or delete any object they must use the controller API.

Controller service provides the following APIs to allow users to manipulate objects:

- network: URL (/networks/) for actions related to network objects.

- vm: URL (/vms/) for actions related to vm objects.

When a request is received, the controller first validates the token to make sure the user is part of the specified workspace and has the permissions required to perform the specified action.

After token validation, the controller validates the request information for example, if the request is to create a new network, the controller first validates that there is no networks with same name in the specified workspace. The validation is performed using GET requests on the URL of the specified object and checking the status code.

Finally, the controller dispatches a task to the corresponding queue (<vm/network>).

2- Queue Monitoring:

Automation services which consume the controller generated tasks, respond with notification message after finishing the task (whether it was successful or not) on a queue named (<vm/network>_notification).

Depending on the content of the notification message, specifically (status: <failed/successful>, retires: <int>), the controller opens a websocket connection to the workspace notification room with superuser token and sends a notification.

3- Retry & Rollback:

If the received notification message has a ‘failed’ status, the controller checks the value of ‘retries’ of that message. If the value is larger than 0, then the controller decrements that value and resubmits the task to the queue. Else, the controller initiates a rollback for that task and sends a notification to the dashboard with task failure.

Components:

1- ControllerAPI:

a REST API which handles user requests from the dashboard or any scripting/automation tool.

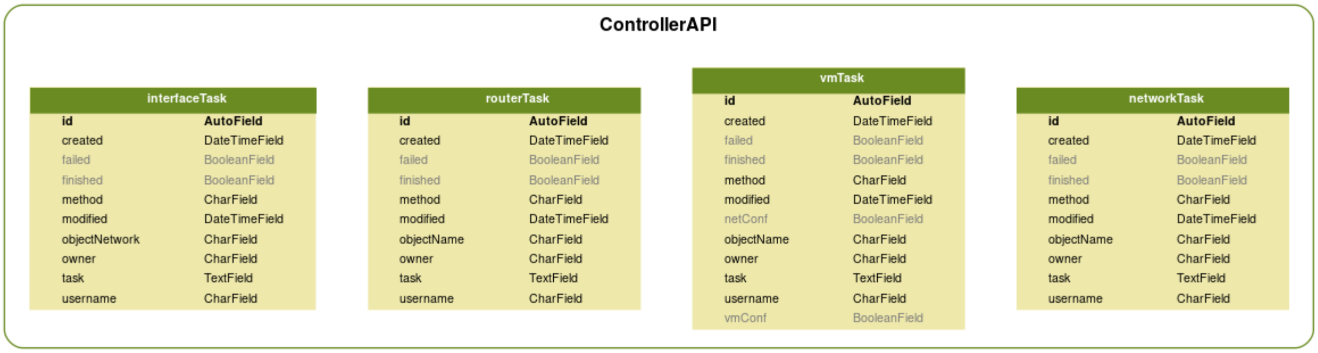
2- Network Notification Consumer:

a daemon process which consumes notification messages generated by network services, and sends notifications.

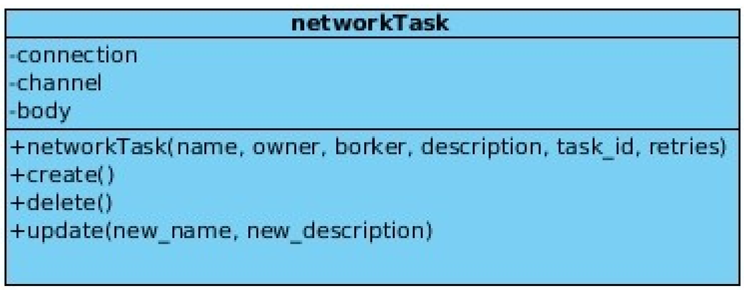
3- VM Notification Consumer:

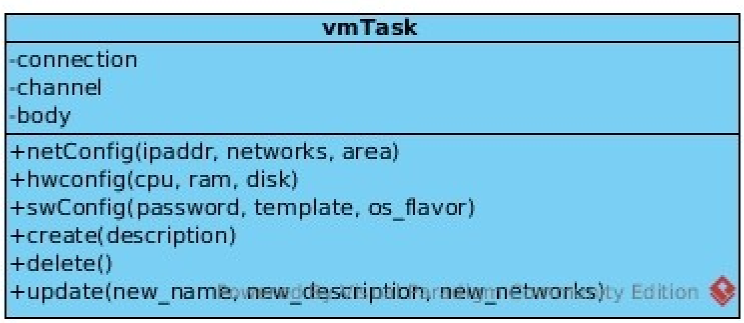
a daemon process which consumes notification messages generated by compute services, and sends notifications.

ERD:



Class Diagram:





8- Notification

Roles:

1- Workspace Notification Rooms:

The notification service is basically a multi-room chat application built using django channels with redis as channels backend. With each workspace represented as a room.

2- Handling End-Users Connections:

Users connect to their workspace room to subscribe for notifications on the URL (ws://<notification>:3000/<workspace name>/<token>).

3- Handling Superuser Connections:

To send notifications, the controller connects to a room with the superuser token and then sends the message.

Only connections with superuser token can send messages to the members of the room.

9- Monitoring

Roles:

1- Monitoring Hosts & Pools:

The main task of the service is to monitor the state of hosts and pools and report any failure to restart the VMs. In case of a host failure (couldn’t connect to the host), the service pushes a message containing the host name to the host_failure queue which will be consumed by compute service to initiate VM restarts on other hosts.

2- Collecting Resources Usage Data:

Another important task, is collecting resources usage memory, cpu and disk (in case of pools) and store them in a database. This information is later used to determine the best way to place a VM on a host or a pool depending on the VM resources.

Componenets:

1- Celery Workers:

They are worker processes that contain the actual monitoring and logging code which is presented as remote functions to be invoked remotely. A leader process divides the hosts in the database to equal groups, which are then sent as parameters to celery workers.

```python
#!usr/bin/python3.6


import libvirt
from config import get_config
from orm_schema import Area, Host, Pool, VirtualMachine
from orm_schema import Host_Stats, Pool_Stats, VirtualMachine_Stats
from orm_io import dbIO
import datetime
from reporters import host_failure
from threading import Thread
from kazoo.client import KazooClient
from celery import Celery
```

```python
app = Celery('tasks', broker='redis://guest@localhost//')
app.conf.broker_url = 'redis://localhost:6379/0'

conf_dict = get_config('conf.json')
db = conf_dict['database']
broker = conf_dict['broker']


def dom_log(body):
	io = dbIO(db)
	vm_stat = VirtualMachine_Stats(
		vm_name=body['name'],
		vm_actual_memory=body['memory_stats']['actual'],
		vm_cpu_time=body['cpu_time'],
		vm_system_time=body['system_time'],
		vm_user_time=body['user_time'],
		vm_available_memory=body['memory_stats']['available'],
		vm_unused_memory=body['memory_stats']['unused']
		)
	io.add(objs=[vm_stat])


def host_log(body):
	io=dbIO(db)
	host_stat = Host_Stats(
		host_name=body['name'],
		host_cpus=body['cpus'],
		host_memory=body['memory'],
		host_free_memory=body['free_memory'],
		state=True
		)
	io.add([host_stat])
	hs = io.query(Host, host_name=body['name'])
	if len(hs) == 0:
		hs = io.query(Host, host_ip=body['name'])
		io.update(
			hs[0], {
					'host_memory': body['memory'],
					'host_free_memory': body['free_memory']
				}
			)
```

```python
def pool_log(body):
	io = dbIO(db)
	pool_stat = Pool_Stats(
		pool_name=body['name'],
		pool_size=body['size'],
		pool_free_size=body['free_size']
		)
	io.add([pool_stat])
	p = io.query(Pool, pool_name=body['name'])[0]
	io.update(p, {'pool_size': body['size'], 'pool_free_size': body['free_size']})


@app.task
def compute_monitor(host):
	io = dbIO(db)
	h = io.query(Host, host_name=host)[0]
	a = io.query(Area, area_id=h.area_id)[0]
	try:
		conn = libvirt.open(f'qemu+ssh://root@{host}/system')
	except Exception as e:
		host_failure(host, a.area_name, broker)
		io.update(h, {'state': False})
		host_stat = Host_Stats(
			host_name=host,
			host_cpus=h.host_cpus,
			host_memory=h.host_memory,
			host_free_memory=h.host_free_memory,
			state=h.state
			)
		io.add([host_stat])
		return 0
	nodeinfo = conn.getInfo()
	host_body = {}
	host_body['name'] = host
	host_body['memory'] = nodeinfo[1] / 1024.0
	host_body['cpus'] = nodeinfo[2]
	host_body['free_memory'] = conn.getFreeMemory() / (1024.0 * 1024.0 * 1024.0)
	host_log(host_body)
	doms = conn.listAllDomains()
	if len(doms) > 0:
		for dom in doms:
		dom_body = {}
		if dom.isActive():
			dom_body['name'] = dom.name()
```

```
                stats = dom.getCPUStats(True)
                dom_body['cpu_time'] = stats[0]['cpu_time']
                dom_body['system_time'] = stats[0]['system_time']
                dom_body['user_time'] = stats[0]['user_time']
                dom_body['memory_stats'] = dom.memoryStats()
                dom_log(dom_body)
        conn.close()


@app.task
def pool_monitor(host):
        conn = libvirt.open(f'qemu+ssh://root@{host}/system')
        pools = conn.listAllStoragePools()
        if len(pools) > 0:
                pool_body = {}
                for pool in pools:
                        info = pool.info()
                        pool_body['name'] = pool.name()
                        pool_body['size'] = info[1] / (1024 * 1024 * 1024)
                        pool_body['free_size'] = info[3] / (1024 * 1024 * 1024)
                        pool_log(pool_body)
        conn.close()
```

2- Zookeeper Clients

Having a leader process to divide the tasks and assigns them to workers equally helps in load balancing and avoids conflicts but, it also introduces a single point of failure. For that reason we are using zookeeper as a middle-ware to elect a process among multiple clients, and to monitor the clients to restart the election process in case the leader fails.

3- RPC Servers

Besides the celery workers and zookeeper clients, there are independent processes that require no coordination and can run in parallel as easily as starting the process. They are the RPC servers which respond to compute service calls to determine what host and pool to place a VM on. They are bulid using the RPC services provided by RabbitMQ.

```
def choose_Host(cpu, memory, area):
	postgres_db = {'drivername': 'postgres',
		'username': 'mon_admin',
		'password': 'Maglab123!',
		'host': db,
		'port': 5432,
		'database': 'monitor'}
	uri = URL(**postgres_db)
	engine = create_engine(uri)
	Session = sessionmaker(bind=engine)
	session = Session()
	io = dbIO(db)
	a = io.query(Area, area_name=area)[0]
	m = memory
	q = session.query(Host).filter(Host.host_free_memory>=m, Host.state==True,
	Host.area_id==a.area_id).all()
	max_m = 0
	max_h = None
	for h in q:
		if h.host_memory >= max_m:
			max_m = h.host_memory
			max_h = h
	return max_h


def choose_Pool(size, area):
	postgres_db = {'drivername': 'postgres',
		'username': 'mon_admin',
		'password': 'Maglab123!',
		'host': db,
		'port': 5432,
		'database': 'monitor'}
	uri = URL(**postgres_db)
	engine = create_engine(uri)
	Session = sessionmaker(bind=engine)
	session = Session()
	io = dbIO(db)
	a = io.query(Area, area_name=area)[0]
	s = size
	q = session.query(Pool).filter(Pool.pool_free_size>=s, Pool.area_id==a.area_id).all()
	max_s = 0
	max_p = None
```

```python
        for p in q:
            if p.pool_size >= max_s:
                max_s = p.pool_size
                max_p = p
        return max_p



def host_request(ch, method, props, body):
        body_dict = json.loads(body.decode('utf-8'))
        h = choose_Host(body_dict['cpu'], body_dict['memory'], body_dict['area'])
        response = {'name': h.host_name, 'ip': h.host_ip}
        jres = json.dumps(response)
        ch.basic_publish(exchange='',
            routing_key=props.reply_to,
            properties=pika.BasicProperties(correlation_id=props.correlation_id),
            body=jres
        )
        ch.basic_ack(delivery_tag=method.delivery_tag)


def pool_request(ch, method, props, body):
        body_dict = json.loads(body.decode('utf-8'))
        p = choose_Pool(body_dict['size'], body_dict['area'])
        response = {'name': p.pool_name}
        jres = json.dumps(response)
        ch.basic_publish(exchange='',
            routing_key=props.reply_to,
            properties=pika.BasicProperties(correlation_id=props.correlation_id),
            body=jres
        )
        ch.basic_ack(delivery_tag=method.delivery_tag)
```

10- Compute

Roles:

1- VM Creation & Deletion:

As the name suggests, the compute service handles the life-cycle of virtual compute resources. Virtual machines are represented as an XML file that describes the hardware of it. For example:

```
<domain type='kvm'>
	<name>kvm-2</name>
	<memory>4194304</memory>
	<currentMemory>4194304</currentMemory>
	<vcpu>2</vcpu>
	<os>
		<type>hvm</type>
	</os>
	<features>
		<acpi/>
		<apic/>
		<pae/>
	</features>
	<clock offset='localtime'/>
	<on_poweroff>destroy</on_poweroff>
	<on_reboot>destroy</on_reboot>
	<on_crash>destroy</on_crash>
	<devices>
		<disk type='file' device='disk'>
			<source file='/home/maged/ISOs/kvm2.qcow2'/>
			<target dev='hda' bus='ide'/>
		</disk>
```

```
		<disk type='file' device='cdrom'>
			<driver name='qemu' type='raw'/>
			<source file='/home/maged/ISOs/Fedora-Server-dvd-x86_64-27-
1.6.iso'/>
			<target dev='hdb' bus='ide'/>
			<readonly/>
			<address type='drive' controller='0' bus='1' unit='0'/>
		</disk>
		<interface type='ethernet'>
			<mac address='00:00:00:00:00:02'/>
			<target dev='kvm2-mgmt' />
		</interface>
		<interface type='ethernet'>
			<mac address='00:10:00:00:00:02'/>
				<target dev='kvm2-pvt' />
		</interface>
		<interface type='ethernet'>
			<mac address='00:20:00:00:00:02'/>
			<target dev='kvm2-pub' />
		</interface>
		<graphics type='vnc' port='5900' autoport='yes' listen='0.0.0.0' />
	</devices>
</domain>
```

In order to create a VM, the compute service first creates the virtual hardware using different classes defined in ‘lib2’ (‘lib’ contains classes for basic management of the virtual hardware but without database operations) and defines an XML description of the VM and then creates the VM using Libvirt.

2- Generating Network Tasks:

After creating the virtual machine and starting it, it’s private network adapter needs to be connected to the specified network. Thus, the compute service sends a message over ‘network’ queue to the network servcie of type:’vm’ that contains (host name, [(network device name, mac address, network), …]).

The network service then uses this information to configure the devices.

Components:

1- VM Worker:

The VM worker is a RabbitMQ consumer process listening to ‘vm’ queue and handles the tasks on that queue. The tasks are mainly create and delete tasks represented by the value of ‘method’ in message body which can be ‘POST’ or ‘DELETE’.

```
#!/bin/python3.6

import pika
import json
from config import get_config
from handlers import vmHandler

conf_file = 'conf.json'
conf_dict = get_config(conf_file)

def method_mapper(method, handler):
        method_dict = {
                'POST': handler.post,
                'PUT': handler.put,
                'DELETE': handler.delete
                }
        return method_dict[method]
```

```
def handler_mapper(t):
	handler_dict = {
		'vm': vmHandler,
		}
	return handler_dict[t]


def consumer(ch, method, properties, body):
	data = json.loads(body.decode('utf-8'))
	t = data['type']
	handler = handler_mapper(t)
	handler.set_config(conf_dict['database'], conf_dict['broker'])
	method = method_mapper(data['method'], handler)
	method(data)


def main():
	connection =
	pika.BlockingConnection(pika.ConnectionParameters(host=conf_dict['broker']))
	channel = connection.channel()
	channel.queue_declare(queue='vm')
	print("handling connection")
	channel.basic_consume(consumer, queue='vm', no_ack=False)
	channel.start_consuming()


if __name__ == '__main__':
	try:
		main()
	except KeyboardInterrupt:
		exit(1)
```

2- Rollback Worker:

Rollbacks have a separate message queue and thus separate listeners. The rollback worker consumes rollback requests concerning compute resources and execute the rollback routine.

```
def handler(ch, method, properties, body):
	data = json.loads(body.decode('utf-8'))
	name = data['name']
	owner = data['owner']
	v = vm().get(name=name, owner=owner)
	flag_dict = v.delete()
	flags = []
	for flag in flag_dict.keys():
		if flag_dict[flag]:
			flags.append(flag)
	if len(flags) > 0:
		line = f"method={data['method']},owner={owner},vm={name},flags={f for f
			in flags}"
		#log line
```

3- Fail Worker:

This worker handles host fail messages sent by the monitoring service. It uses a special method in the 'lib2.computeOps.vm' class build to handle such case.

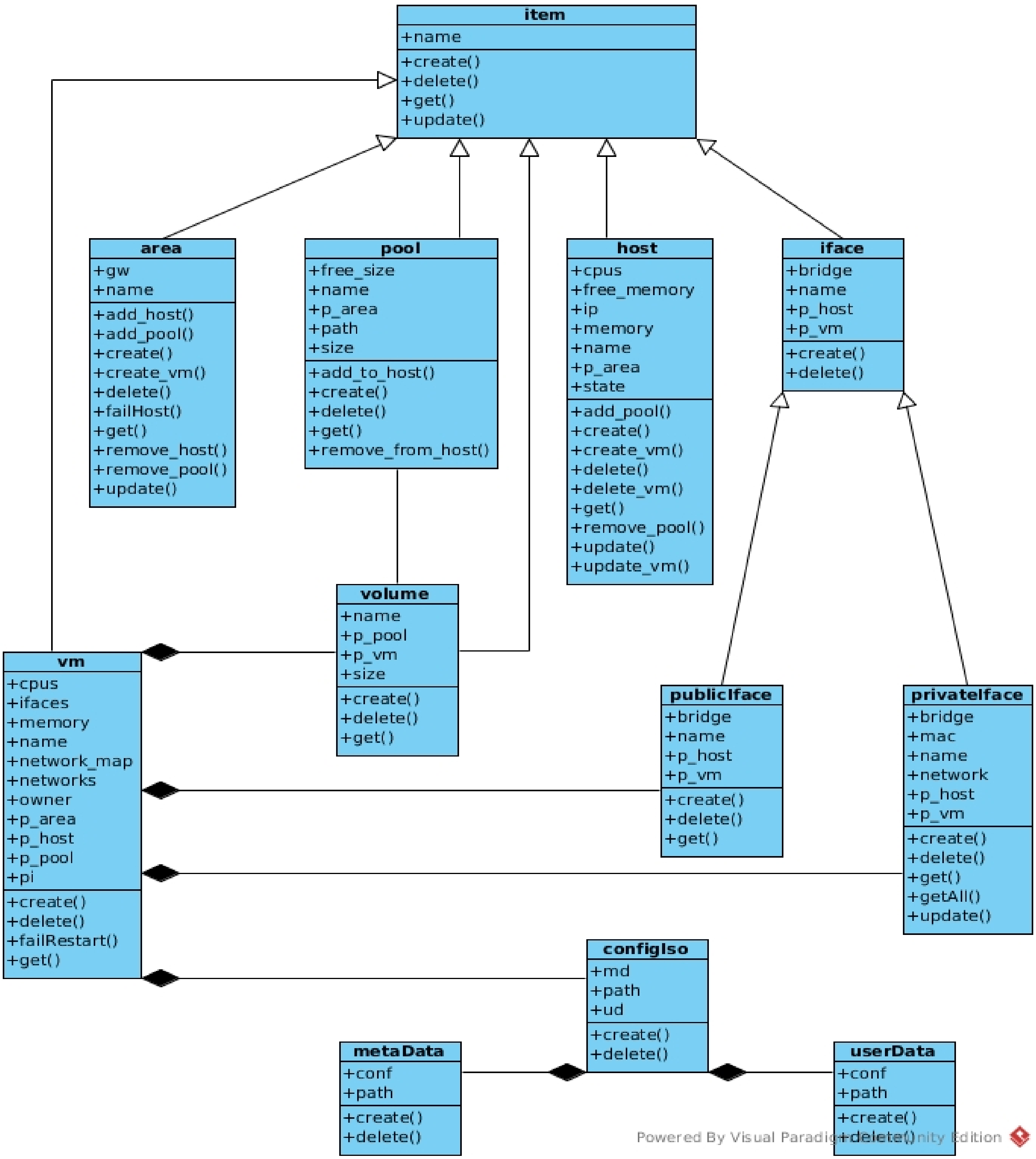
```
def failRestart(self, new_host):

	self.p_host = new_host
	io = dbIO(database)
	a = io.query(Area, area_name=self.p_area.name)[0]
	h = io.query(Host, host_name=new_host.name, area_id=a.area_id)[0]
	v = io.query(VirtualMachine, vm_name=self.name, vm_owner=self.owner)
	io.update(v, {'state': False, 'host_id': h.host_id})
	pi = publicIface(self, self.p_host)
	try:
		pi.create()
		self.pi = pi
	except Exception as e:
```

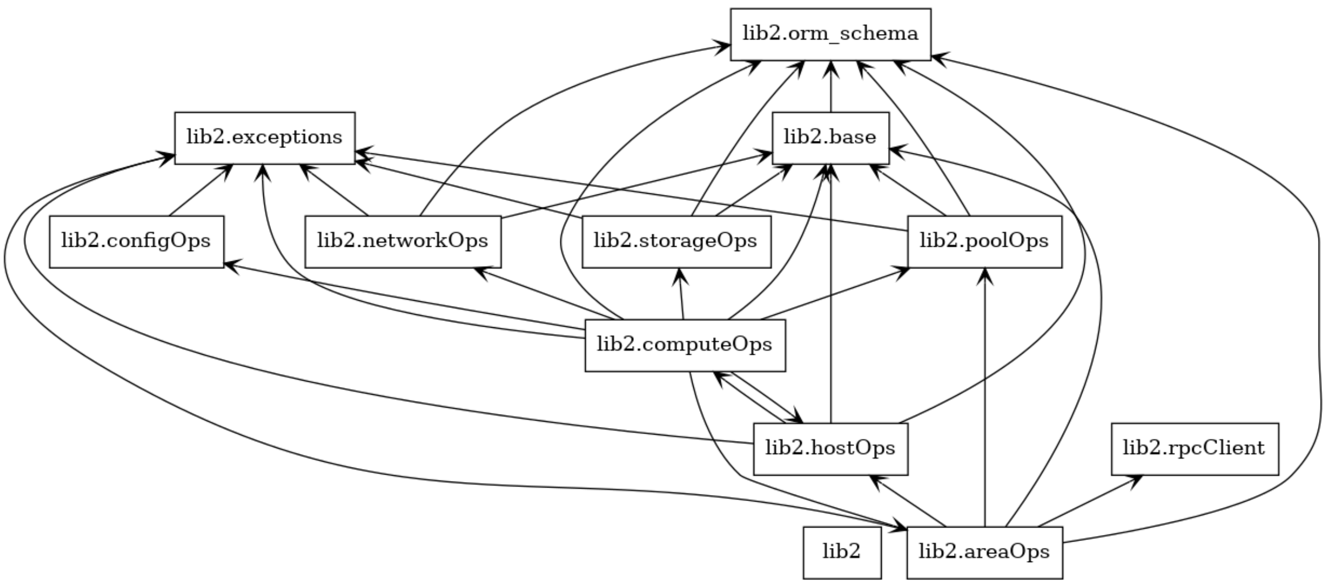
```
            raise CreateVmException('failed to create public interface')
    network_XML = ''
    self.network_map = []
    if len(self.ifaces) > 0:
        for iface in self.ifaces:
            try:
                m = self._genMacAddr()
                pvi = privateIface(self, self.p_host, iface.network, iface.mac)
                pvi.create()
                self.ifaces.append(pvi)
                map_dict = {}
                map_dict['name'] = pvi.name
                map_dict['host'] = self.p_host.name
                map_dict['network'] = iface.network
                map_dict['mac'] = m
                self.network_map.append(map_dict)
                network_XML += f'''<interface type="ethernet">
                    <mac address="{m}"/>
                    <target dev="{pvi.name}"/>
                    </interface>'''
            except Exception as e:
                pi.delete()
                if len(self.ifaces) > 0:
                    for i in self.ifaces:
                        i.delete()
                raise CreateVmException('failed to create private interfaces')
    vol = io.query(Volume, vm_id=v.vm_id)[0]

    XMLConf = f'''
            .....
'''
    try:
        conn = libvirt.open(f"qemu+ssh://root@{self.p_host.ip}/system")
        dom = conn.defineXML(XMLConf)
        dom.create()
    except Exception as e:
        pi.delete()
        if len(self.ifaces) > 0:
            for i in self.ifaces:
                i.delete()
        io.delete([v])
        raise CreateVmException('failed to create vm')
    conn.close()
    io.update(v, {'state': True})
```

Class Diagram:



Package Diagram:



11- Network

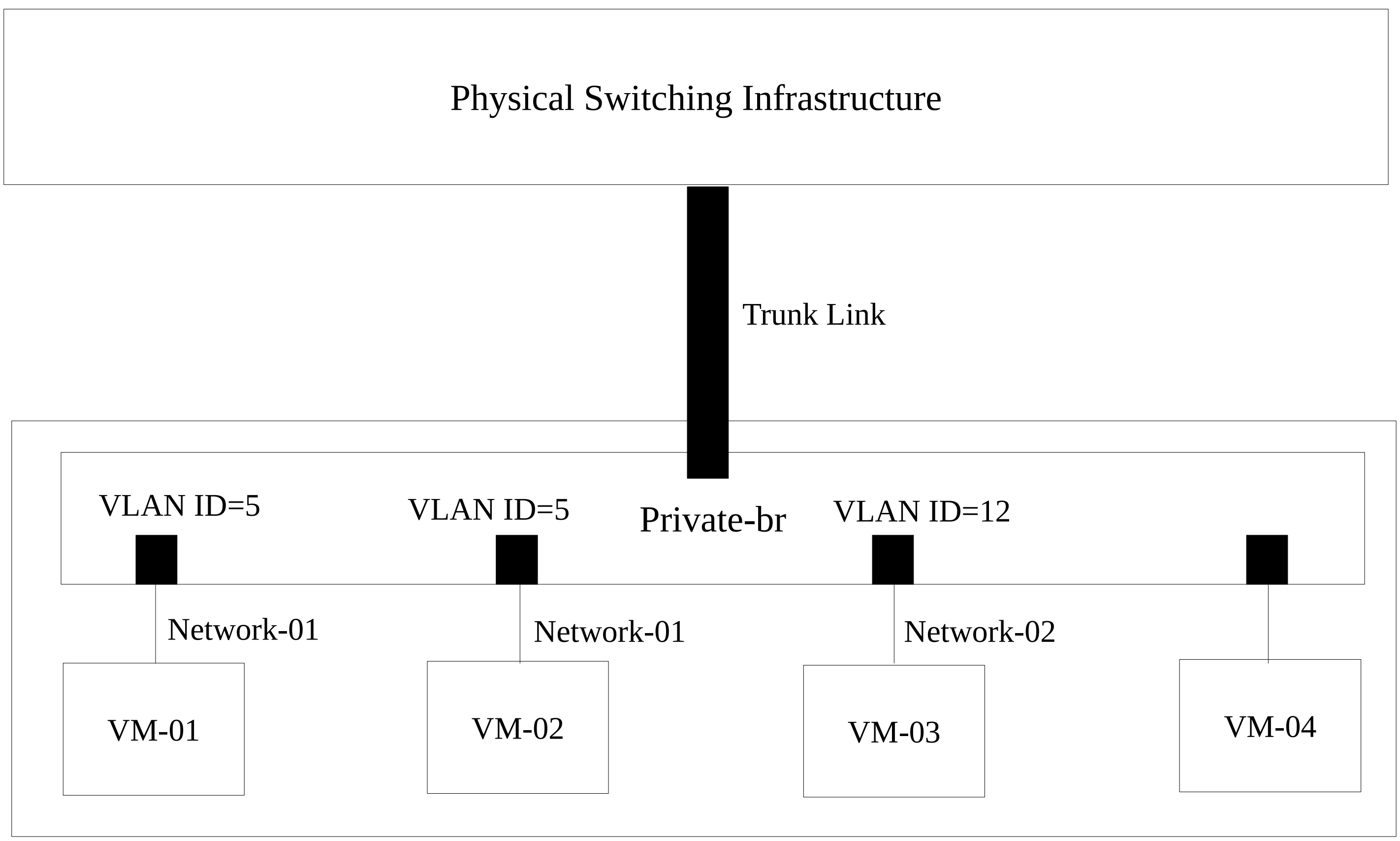
Roles:

1- Network Creation & Deletion:

Currently, we are using VLANs to isolate virtual networks. Each virtual network is assigned a unique VLAN ID from 1 to 4095 (except for reserved IDs).

2- Managing Interfaces:

Later any network device connected to that network is tagged with it's associated VLAN ID.



Components:

1- Network Worker:

It is a RabbitMQ consumer that's responsible for creating and deleting networks as well as configuring VM interfaces.

2- Rollback Worker:

The rollback worker consumes rollback requests concerning network resources and execute the rollback routine.

(to handle interface configuration we built a dedicated class 'lib.NetworkOps.Interface')

```
class Interface:

    def __init__(self, name='', network=None, host='', mac=''):
        self.name = name
        self.network = network
        self.host = host
        self.mac = mac

    def create(self):
        io = dbIO(database)
        n = io.query(Network, network_name=self.network.name,
        network_owner=self.network.owner)[0]
        h = io.query(Host, host_name=self.host)[0]
        ssh = paramiko.SSHClient()
        ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        ssh.connect(h.host_ip, username='root')
        cmd1 = f'ovs-vsctl set Port {self.name} tag={n.vlan_id}'
        stdin, stdout, stderr = ssh.exec_command(cmd1)
        stdin.close()
        e = stderr.read()
        if e:
            print(e)
            ssh.close()
            raise Exception(e)
        ssh.close()
```

```
        iface = Iface(
                iface_name = self.name,
                iface_mac = self.mac,
                network_id = n.network_id,
                host_id = h.host_id
                )
        io.add([iface])

    @classmethod
    def get(cls, name, network):
        io = dbIO(database)
        n = io.query(Network, network_name=network.name,
                network_owner=network.owner)[0]
        Ifaces = io.query(Iface, iface_name=name, network_id=n.network_id)
        if len(ifaces) == 0:
                return None
        iface = ifaces[0]
                h = io.query(Host, host_id=iface.host_id)[0]
        return cls(
                name,
                network,
                h.host_name,
                iface.iface_mac
                )

    def delete(self):
        io = dbIO(database)
        iface = io.query(Iface, iface_name=self.name, iface_mac=self.mac)[0]
        io.delete([iface])
```

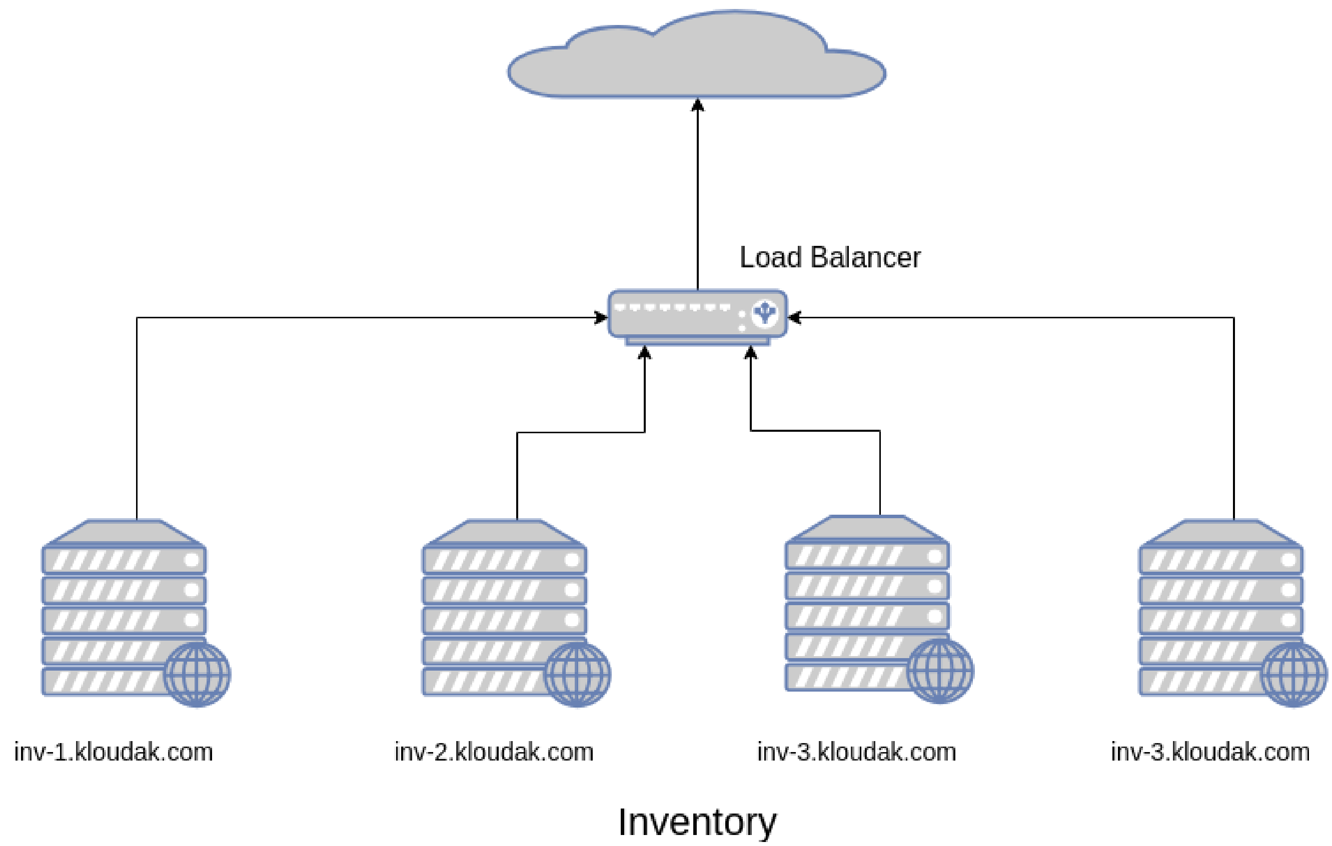
12- Availability, Reliability & Coordination

Availability:

1- Multiple Instances:

Kloudak services are designed in a way to make them run in parallel with no extra configuration (except for monitoring service) just as simple as starting the process.

For example, you can deploy inventory (and all data services) like this:



For automation services, you can start multiple instances of any process with no more configuration than the previously mentioned in *chapter 4: Deployment Guide.*

2- Infrastructure Monitoring & Recovery:

To deal with host failures in the infrastructure, we built a monitoring service to report any failures and built special handlers in other services to restart virtual machines again.

Of course, this still has the downside of down-time until restart.

Reliability:

1- Handling Connection Failures:

In case of a connection failure, the requesting service waits a period and then retries again. The number of retries and the time between retries is added to the configuration files of all services.

2- Handling Service Failures:

In case of an automation service failure, the task it was handling would simply be re-queued and consumed by another service.
Data services failures wouldn't likely to cause a problem for end users as the controller submits tasks to automation services before it updates the inventory. It would give a false notification or create a resource without being traced in the inventory service.

3- Handling Execution Errors & Rollbacks:

In case of an an execution error (no matter the reason), the service will respond with a failed notification message on the corresponding notification queue. The controller 'QueueMonitoring' processes will consume those notifications and re-submit the task if it hasn't exceeded the number of retries. If the task has 'retries' of 0, the controller will initiate a rollback for that task.

rollback.py

```
#!/usr/bin/python3.6

import json, pika
import helpers
from tasks import dispatch

def vmRollback(task, inventory, broker, body={}):
	from helpers import api_call
	'''
	if method == POST:
		1- delete the entry in inventory
		2- if vm_notification:
			delete network entries in network service
		else:
			delete vm from compute service
	if method == DELETE:
		add entry to inventory
		log task
	'''
	owner = task.owner
	name = task.objectName
	body = {
		'owner': owner,
		'name': name,
		'method': task.method,
		'type': 'vm'
		}
	if task.method == 'POST':
		if 'host' in body.keys():
			#failed at network config. delete vm
			dispatch(json.dumps(body), 'vm_rollback', broker)
		else:
			#delete interfaces at network service
			dispatch(json.dumps(body), 'network_rollback', broker)
			url = f'{inventory}{owner}/vms/{name}/'
			api_call('delete', url)
	elif task.method == 'DELETE':
		#add log entry
		pass
```

```python
def networkRollback(task, inventory, borker):
        from helpers import api_call
        """
        1- if method == POST:
                delete the entry in inventory
        if method == DELETE:
                add entry to inventory
        """
        owner = task.owner
        name = task.objectName
        if task.method == 'POST':
                url = f'{inventory}{owner}/networks/{name}/'
                api_call('delete', url)
        elif task.method == 'DELETE':
                #add log entry
                url = f'{inventory}{owner}/networks/'
                body = {
                        'name': name,
                        'description': ''
                        }
                api_call('post', url, body)
```

Coordination:

1- Monitoring Coordination:

Coordination in the monitoring services is achieved in two steps:

First, electing a leader process to be responsible for distributing tasks among available processes.
Every zookeeper client is assigned a numeric unique id at the start of the session. The client process with the minimum id is elected as leader.

Second, the leader fetches host information from the database and sends them to available celery workers evenly.

(in case the leader fails, its id is removed and the election process is re-initiated)

Appendix: System APIs

Inventory API:

Workspace Objects:

URL: <IP>:<Port>/workspaces/

Methods:

```
- GET (retrieves all workspaces)
- POST (creates a workspace object)
```

Body:

{

"name": "Workspace Name"

}

Response:

{

“workspaces”: [{"name": "Workspace Name"},]

}

Status Codes:

201 = Successful POST

200 = Successful GET

400 = Error

URL: <IP>:<Port>/workspaces/<workspace>/

Methods:

```
- GET (retrieves workspace information)
- DELETE (deletes a workspace object and all objects in that workspace)
- PUT (updates workspace information)
```

Body:

{

"name": "Workspace Name"

}

Infrstructure Objects:

URL: <IP>:<Port>/<workspace>/vms/

Methods:

- GET (retrieves all of user’s virtual machines)
- POST (creates virtual machine object)

Body:

{

"name": "VM Name",

"description": "VM Description",

"ip": "10.10.10.3/24",

"state": "U",

"area": "Area Name",

"template": "Template Name",

"networks": ["Network-01 Name", "Network-02 Name"]

}

Response:

{

“vms”: [{"name": “VM Name"},]

}

states = [('C_D', 'Creating_Disk'), ('C_N', 'Configuring_Network'), ('U', 'UP'), ('D', 'Down')]

URL: <IP>:<Port>/<workspace>/networks/

Methods:

- GET (retrieves all of user’s networks)
- POST (creates network object)

Body:

{

"name": "Network Name",

"description": "Network Description",

"state": "U"

}

Response:

{

“networks”: [{"name": “Network Name"},]

}

states = [('U', 'Up'), ('C', 'Creating')]

URL: <IP>:<Port>/<workspace>/vms/<vm>/

Methods:

- GET (retrieves all information of a virtual machine)
- DELETE (deletes a virtual machine object)
- PUT (updates virtual machine information)

Body:

{

"name": "VM Name",

"description": "VM Description",

"networks": [{“name”: "Network-01 Name"},],

"state": "U"

}

states = [('C_D', 'Creating_Disk'), ('C_N', 'Configuring_Network'), ('U', 'UP'), ('D', 'Down')]

URL: <IP>:<Port>/<workspace>/networks/<network>

Methods:

- GET (retrieves all information of a network)
- DELETE (deletes a network object)
- PUT (updates network information)

Body:

{

"name": "Network Name",

"description": "Network Description",

"state": "U"

}

states = [('U', 'Up'), ('C', 'Creating')]

URL: <IP>:<Port>/<workspace>/routers/

Methods:

- GET (retrieves all of user’s routers)
- POST (creates router object)

Body:

```
{
    "name": "Network Name"
}
```

Response:

```
{
    “routers”: [{"name": “Router Name"},]
}
```

URL: <IP>:<Port>/<workspace>/routers/<router>

Methods:

- GET (retrieves all information of a router)
- DELETE (deletes a router object)
- PUT (updates router information)

Body:

```
{
    "name": "Router Name"
}
```

URL: <IP>:<Port>/<workspace>/routers/<router>/interfaces/

Methods:

- GET (retrieves all of routers’ interfaces)
- POST (creates a router interface)

Body:

```
{
    "network": "Network-01",
    “ip”: “192.168.1.1/24”
}
```

Response:

```
{
    “interfaces”: [{"name": “Interface Name"},]
}
```

URL: <IP>:<Port>/<workspace>/routers/<router>/interfaces/<interface network>

Methods:

- GET (retrieves all information of a router interface)
- DELETE (deletes a router interface object)
- PUT (updates router interface information)

Body:

```
{
    "ip": "10.10.10.1/24"
}
```

Note:

Manipulating inventory objects (except Workspace and User) must be done via the controller service.

Template Objects:

URL: <IP>:<Port>/templates/

Methods:

```
- GET (retrieves all templates)
- POST (creates a template object)
```

Body:

{

"name": "Template Name",

"description": "Template Description",

"os": "Template OS",

"cpu": int,

"ram": int,

"disk": int

}

Response:

{

“templates”: [{"name": “Template Name"},]

}

cpu = 2 (cores)   ram = 2 (GiB)   disk = 40 (GiB)

URL: <IP>:<Port>/templates/<template>/

Methods:

```
- GET (retrieves template information)
- DELETE (deletes a template object)
- PUT (updates a template object)
```

Body:

{

"name": "Template Name",

"description": "Template Description",

"os": "Template OS",

"cpu": int,

"ram": int,

"disk": int

}

cpu = 2 (cores) ram = 2 (GiB) disk = 40 (GiB)

Area Objects:

URL: <IP>:<Port>/areas/

Methods:

```
- GET (retrieves all areas)
- POST (creates an area object)
```

Body:

{

"name": "Area Name",

"subnet": "10.10.10.0/24",

"next_ip": "10.10.10.2/24"

}

URL: <IP>:<Port>/areas/<area>/

Methods:

```
- GET (retrieves area information)
- DELETE (deletes an area object)
- PUT (updates area information)
```

Body:

{

"name": "Area Name",

"subnet": "10.10.10.0/24",

"next_ip": "10.10.10.2/24"

}

Response:

{

“areas”: [{"name": "Area Name"},]

}

Address Objects:

URL: <IP>:<Port>/address/<area>/get_ip/

Methods:

```
- GET (returns a free IP address of the area subnet)
```

User Objects:

URL: <IP>:<Port>/<workspace>/users/

Methods:

```
- GET (retrieves all users in a workspace)
- POST (adds a new user to a workspace)
```

Body:

{

"name": "username",

"vm_can_add": True,

"vm_can_edit": False,

"vm_can_delete": False,

"network_can_add": True,

"network_can_edit": False,

"network_can_delete": False,

"router_can_add": True,

"router_can_edit": False,

"router_can_delete": False,

"user_can_add": True,

"user_can_edit": False,

"user_can_delete": False

}

Response:

{

“users”: [{"name": "User Name"},]

}

URL: <IP>:<Port>/<workspace>/users/<username>/

Methods:

```
- GET (retrieves user information)
- DELETE (deletes a user)
- PUT (update user permissions)
```

Body:

{

"vm_can_add": True,

"vm_can_edit": False,

"vm_can_delete": False,

"network_can_add": True,

"network_can_edit": False,

"network_can_delete": False,

"router_can_add": True,

"router_can_edit": False,

"router_can_delete": False,

"user_can_add": True,

"user_can_edit": False,

"user_can_delete": False,

}

Inventory API:

VM Requests:

URL: <IP>:<Port>/vms/

Method:

- POST (VM Creation Request)

Body:

{

"name": "VM Name",

"description": "VM Description",

"owner": "Workspace Name",

"networks": ["Network-01 Name", "Network-02 Name",],

"area": "Area Name",

"template": "Template Name",

"password": "raw password"

}

Response:

{

"name": "VM Name",

"description": "VM Description",

"owner": "Workspace Name",

"networks": ["Network-01 Name", "Network-02 Name",],

"area": "Area Name",

"template": "Template Name",

"password": "raw password",

“ip”: “10.10.10.7/24”

}

URL: <IP>:<Port>/vms/

Method:

- DELETE (VM Deletion Request)

Body:

```
{
    "name": "Network Name",
    "owner": "Workspace Name"
}
```

URL: <IP>:<Port>/vms/

Methods:

- PUT (Edit VM Information)

Body:

```
{
    "name": "VM Name",
    "owner": "Workspace Name",
    "update_dict": {
            "name": "New VM Name",
            "description": "New VM Description",
            "new_networks": [{“name”: "Network-01 Name"},]
            }
}
```

Network Requests:

URL: <IP>:<Port>/networks/

Method:

- POST (Creation Request)

Body:

```
{
    "name": "Network Name",
    "description": "Network Description",
    "owner": "Workspace Name"
}
```

URL: <IP>:<Port>/networks/

Method:

- DELETE (Network Deletion Request)

Body:

```
{
    "name": "Network Name",
    "owner": "Workspace Name"
}
```

URL: <IP>:<Port>/networks/

Methods:

- PUT (Edit Network Information)

Body:

{

"name": "Network Name",

"owner": "Workspace Name",

"update_dict": {

"name": "New Network Name",

"description": "New Network Description"

}

}

Router Requests:

URL: <IP>:<Port>/routers/

Method:

- POST (Router Creation Request)

Body:

{

"name": "Router Name",

"owner": "Workspace Name"

}

URL: <IP>:<Port>/routers/

Method:

- DELETE (Router Deletion Request)

Body:

{

"name": "Router Name",

"owner": "User Name"

}

URL: <IP>:<Port>/routers/

Methods:

- PUT (Edit Network Information)

Body:

```
{
        "name": "router Name",
        "owner": "Workspace Name",
        "update_dict": {
                "name": "New Router Name"
                }
}
```

Router Interface Requests:

URL: <IP>:<Port>/interfaces/

Method:

- POST (Router Interface Creation Request)

Body:

```
{
        "network": "Interface Network",
        "owner": "Workspace Name",
        “router”: “Router Name”,
        “ip”: “10.10.10.1/24”
}
```

URL: <IP>:<Port>/interfaces/

Method:

- DELETE (Router Interface Deletion Request)

Body:

```
{
    "network": "Interface Network",
    "owner": "User Name",
    “router”: “Router Name”
}
```

URL: <IP>:<Port>/interfaces/

Methods:

- PUT (Edit Router Interface Information)

Body:

```
{
    "router": "router Name",
    “network”: “network name”
    "owner": "Workspace Name",
    "update_dict": {
            "ip": "20.10.16.254/24"
            }
}
```

Tasks Requests:

URL: <IP>:<Port>/tasks/<workspace>/

Method:

- GET (retrieves all tasks for a workspace)

URL: <IP>:<Port>/running_tasks/<workspace>/

Method:

- GET (retrieves running tasks of a workspace)

URL: <IP>:<Port>/finished_tasks/<workspace>/

Method:

- GET (retrieves finished tasks of a workspace)