OpenStack Cloud Service Portal

Submitted by: Project Group 3

Table of Contents

1. Introduction 3

2. Objectives 3

3. Architecture 4

4. Design 6

5. Implementation 12

5.1. Overview 12

5.2. Web UI 13

5.2.1. Resource Usage 14

5.2.2. Services 15

5.2.3. Tasks 19

6. Observations and Challenges 19

7. Authors 20

8. Bibliography 20

Table of Figures

Figure 1 - OpenStack architecture 4

Figure 2 - OpenStack Components used in the project 5

Figure 3 - OpenStack APIs Sequence Diagram 7

Figure 4 - Data Model 8

Figure 5 - Data access object layer 9

Figure 6 - OpenStack and SSH Client Layer 9

Figure 7 - OpenStack Server Routes 10

Figure 8 – Asynchronous Job Classes 11

Figure 9 – A Single page application (Web UI) 13

Figure 10 - Resource Usage 14

Figure 11 - Services Table 15

Figure 12 - Add Service 15

Figure 13 - Delete Service 16

Figure 14 - Get Service 17

# Introduction

This document is a report on the implementation of OpenStack based Self-Service Portal project. It describes the functional requirements, design and architecture of the implementation.

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

# Objectives

Using a web UI end user should be able to perform the following on an OpenStack instance:

1. Create multiple VM instances, and the associated networking (interfaces, subnets, router and connect the 2 VMs via the router)
2. The VM image upon bootup should bring up some service like:
   1. a basic web server with some hello world web page hosted on it or
   2. a DB server that should start upon VM boot up
3. There could be 2 types (minimal deliverables):
4. **Basic web site** - which will just create one VM with the web server
5. **Big web site** - which will create multiple VMs each connected in a cluster and each hosting a web application excluding one VM which will host the DB server.
6. They should be named appropriately, “Web VM1”, “Web VM2” and “Database VM”.
7. Orchestration should ensure that DB VM is started first then Web VM 1 and then Web VM 2.
8. We can think of front ending this cluster with load balancer if more than one VM is provisioned (only the web server VMs and not DB)
9. If user specifies a cluster size that cannot be provisioned due to no capacity, then an error should be shown.
10. Any error that shall occur during provisioning should be reported in response on the Web UI.
11. UI should be a Web UI.
12. Backend should use following 2 ways to connect to OpenStack:
    1. OpenStack CLI via SSH to OpenStack VM
    2. OpenStack REST APIs.
13. Additional deliverables:
14. Measure time consumed by each VM and create a usage bill.
15. Every 10,000 CPU seconds cost $0.01 or something similar. We can do so by measuring the uptime of the VM.
    * + 1. Status dashboard showing health of each system component
        2. Show number of resources (cpu, memory, HDD) available.
16. Optional Simulator:
    * + 1. Simulate OpenStack APIs
        2. Return hard coded REST responses
        3. Need to ensure that our project works well with Simulator in the event that the OpenStack installation is not working

# Architecture

This section talks about the architecture of the OpenStack deployment.

Oracle VM Server 3.3.3 will be used for this project to host the OpenStack Icehouse release.

OpenStack will be installed using the packstack utility. Packstack uses puppet configuration management tool to automate the installation of OpenStack on one or more servers.

For this project, Packstack has been used to install all components of OpenStack on a single VM. The VM is hosted on VirtualBox 5.0.2.

|  |
| --- |
| [root@CMPE-OPENSTACK ~(keystone\_admin)]# lsb\_release -a  LSB Version: :base-4.0-amd64:base-4.0-noarch:core-4.0-amd64:core-4.0-noarch:graphics-4.0-amd64:graphics-4.0-noarch:printing-4.0-amd64:printing-4.0-noarch  Distributor ID: OracleVMserver  Description: Oracle VM server release 3.3.3  Release: 3.3.3  Codename: n/a |

The figure below shows the essential components of OpenStack environment. There are pools of compute, network and storage resources which are managed via OpenStack Cloud Operating System to provide cloud based services. The OpenStack Horizon dashboard provides a self-service portal for tenant users to manage the cloud environment.

In this project, our application will use both the REST APIs exposed by OpenStack and OpenStack CLI to create a self-service portal for creating cloud based database and web services.

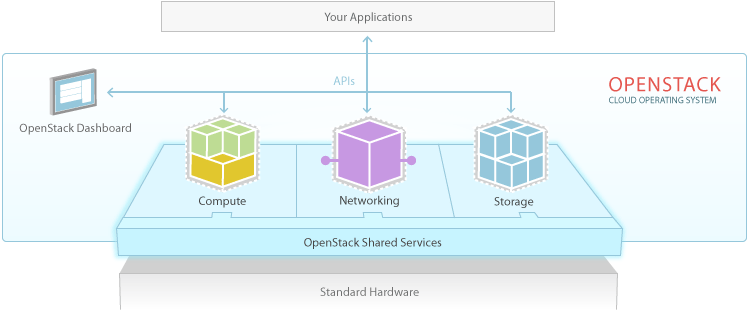


Figure - OpenStack architecture

In this project the following components of OpenStack, shown in the below figure, are used.

1. Nova Compute
2. Neutron Network
3. Glance Images - with file system being used as backing store for VM image files.
4. Keystone Identity
5. Horizon Dashboard - this is present and is only used for verification and presentation of actions performed through API.

All OpenStack components will be co-located on a single VM (which is hosted on VirtualBox). Following configuration will be used for the VM:

1. CPUs = 2
2. Memory = 6500 MB
3. Network = 2 vNICs (both NAT attached)
   1. vNIC1 will be used to administer OpenStack
   2. vNIC2 will be used to access OpenStack dashboard, APIs and CLI (via SSH)

Following configuration will be used for the VMs that OpenStack creates:

1. CPU = 1
2. Memory = 1200 MB
3. Network = vNIC1 connected to single subnet

The OpenStack VM itself will consume the remaining memory of 4.5GB.

The OpenStack VM will have Oracle VM Server 3.3.3 which is a distribution of Xen hypervisor. Xen will be used to provision VMs within the OpenStack VM.

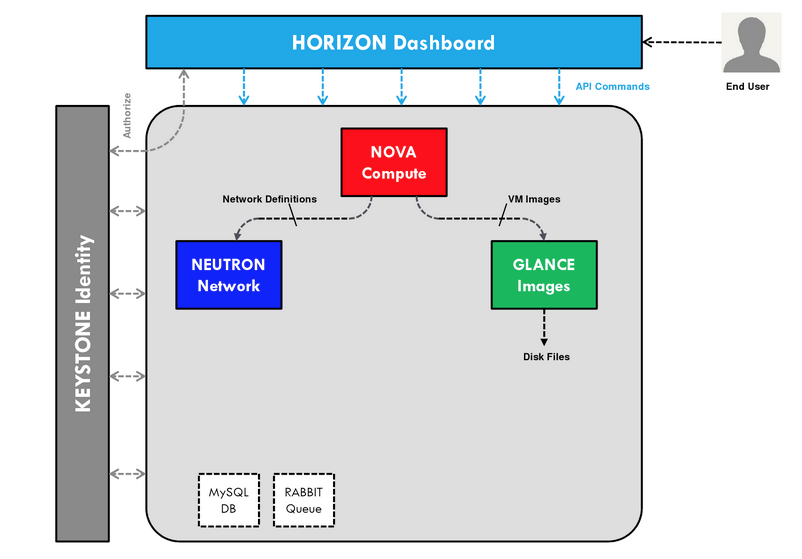


Figure - OpenStack Components used in the project

# Design

OpenStack Service portal presents to a tenant user a Web UI to create 2 types of services:

* 1. **Basic Service** – which defaults to creating just one Web VM.
  2. **Big Service** – where user can specify the number of VMs he will like to have in the service. Of the number specified, 1 VM will be created as a DB VM and N-1 VMs will be created as Web VM.
* Web VM = Ubuntu 14.04 running lighttpd
* DB VM = Ubuntu 14.04 running mongodb

In both the VMs, the service (web server or DB server) will be started upon VM boot strap.

The Service creation, deletion and resource usage collection are performed asynchronously so the UI remains responsive. Service status is updated in the backend eventually when the service operation completes and upon next UI page refresh user can see the updated service/VM status.

Following APIs are provided by the cloud service are consumed by the Web UI:

* 1. Add Service – Add a new service of basic or big type.
  2. Get Service Details – Get information on the service, like the associated VMs, the logs for the service etc.
  3. Delete Service – Delete a service.
  4. Get Tasks – Tasks are events that have occurred in the system asynchronously. A chronological list of such events (or tasks performed) are returned by this API. For example, a VM being created, VM being deleted etc.
  5. Get resource usage – gets the current resource usage per tenant/user combination.

Following sequence diagram shows how the above APIs work in the system:

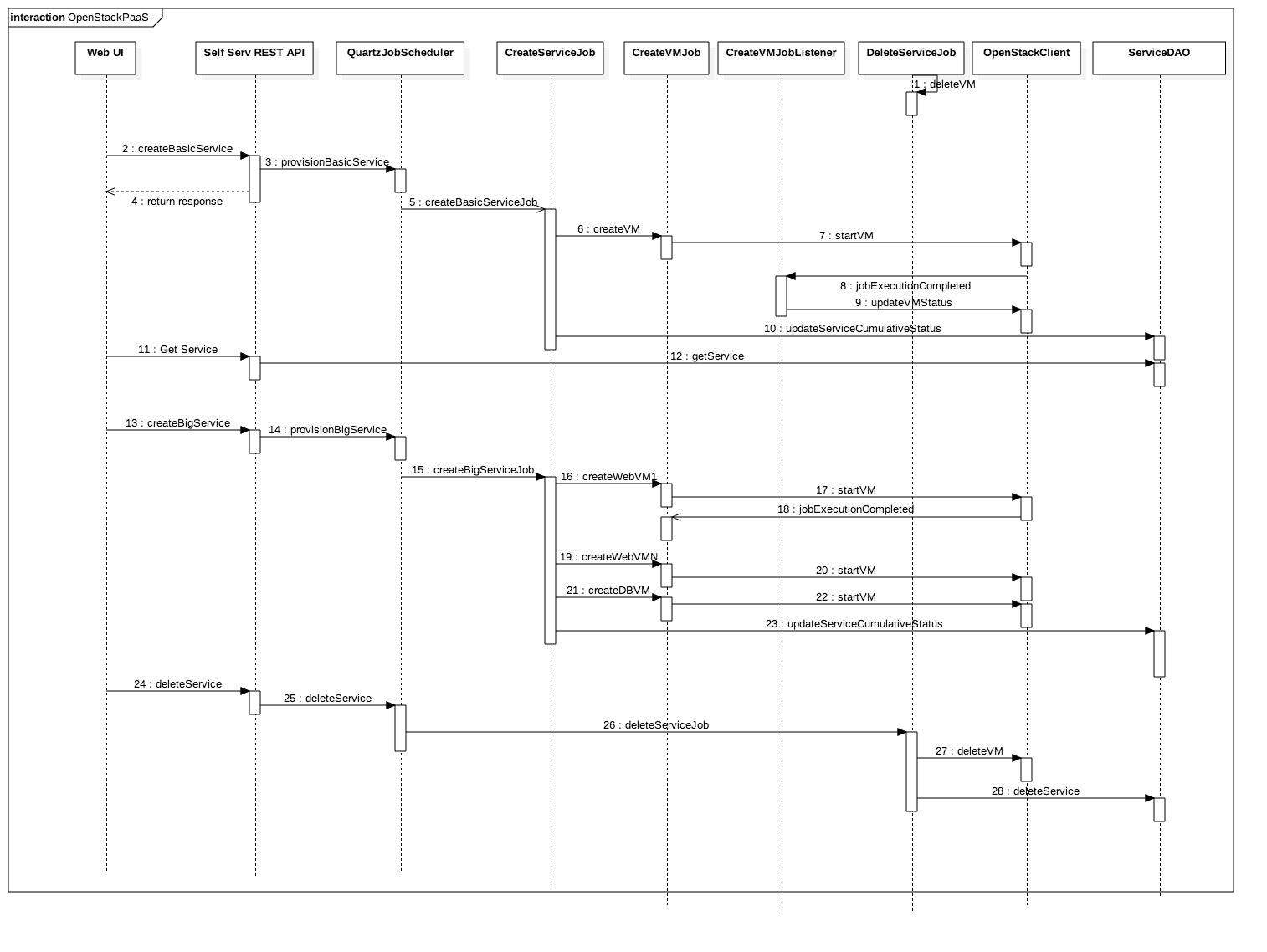


Figure - OpenStack APIs Sequence Diagram

1. Web UI – is the OpenStack Cloud Service portal UI from where user can create services, delete services and check on the provisioning status for a service.
2. REST API – backend exposes REST APIs which is consumed by the UI.
3. Quartz job scheduler – is the job manager framework which is used to schedule create service, create VM, delete service, resource usage collector jobs.
4. Create VM Job – is used to create multiple VMs in parallel asynchronously without requiring the user request to wait for the provisioning of the service to complete. Once all service VMs are provisioned, the service status is updated in DB.
5. Create VM Job Listener – is a callback interface for the job scheduler which gets called back when job completes executing. So when VM is provisioned, the job listener can update the database with the VM status.
6. Delete VM Job – is the job to delete the VM from both the database and OpenStack.
7. OpenStack client – is a layer to wrap the API library being used to interface with OpenStack.
8. ServiceDAO – is one of the data access objects that wrap all interactions with the database layer. Other DAOs in the system are shown in the class diagram below.

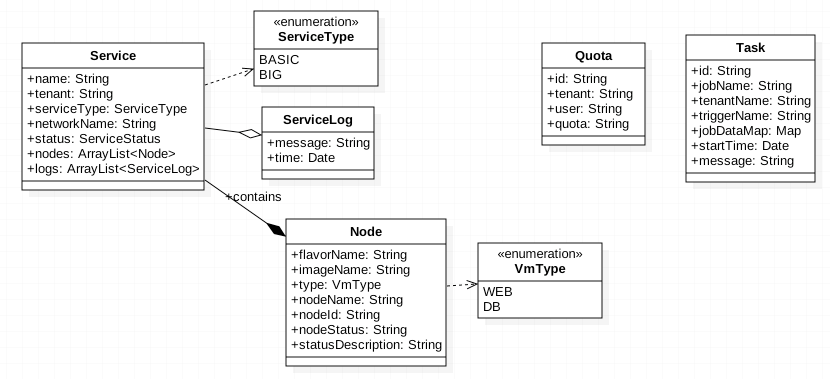


Figure - Data Model

Above class diagram shows the data model for the system.

1. Service – represents the service entity in the DB.
2. Node – represents the VM. A service can have one or more nodes.
3. Quota – is used to persist the resource usage information periodically per tenant/user combination. This is done so that when the UI needs the information it is fetched from the DB rather than from OpenStack. The polling interval is 5 mins.
4. Task – is the table for recording all job executions in the backend.

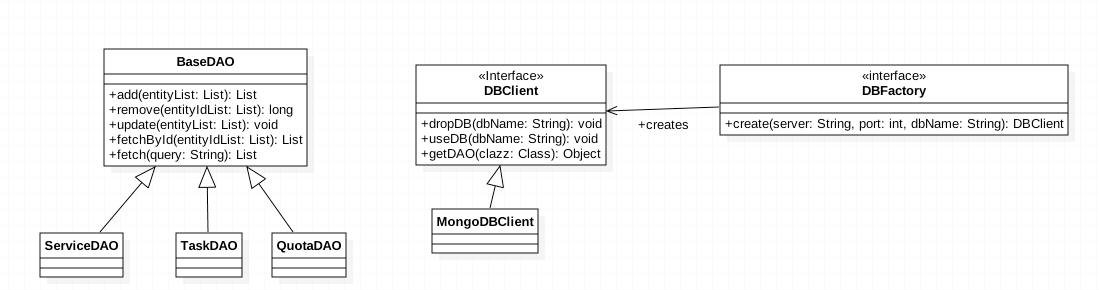


Figure - Data access object layer

The above class diagram is for the database access layer in the system. It shows the 3 main classes:

* 1. Service DAO – for service entity
  2. Task DAO – for task entity
  3. Quota DAO – for resource usage entity
  4. DBClient – provides an abstraction for Mongo DB client.
  5. DBFactory – provides a way to inject a DBClient implementation (which can be a different DB than MongoDB or for testing it could be a Mockup data).

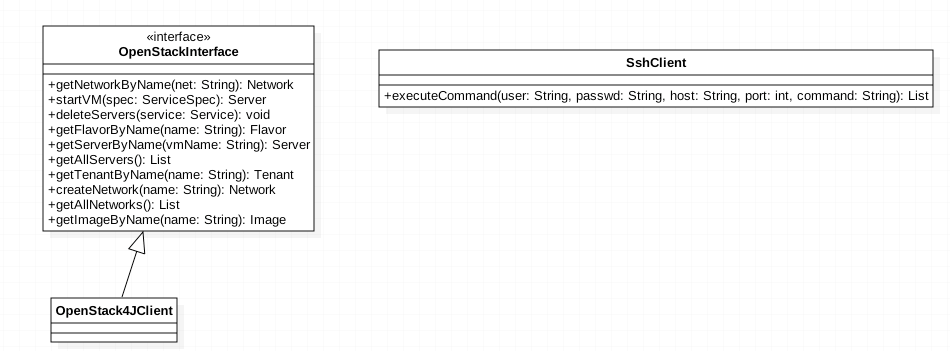


Figure - OpenStack and SSH Client Layer

The above classes are for accessing OpenStack by using OpenStack REST APIs and SSH.

* 1. OpenStack4JClient – is a wrapper implementing the methods declared in OpenStackInterface using the [OpenStack4J](http://openstack4j.com/) library.
  2. OpenStackInterface – declares those methods that are required by the system from OpenStack.
  3. SshClient – is a SSH client which can be used generically to perform ssh operation to any host but in this project it is used to connect to OpenStack VM and to collect the resource usage metrics every 5 mins per tenant/user combination.

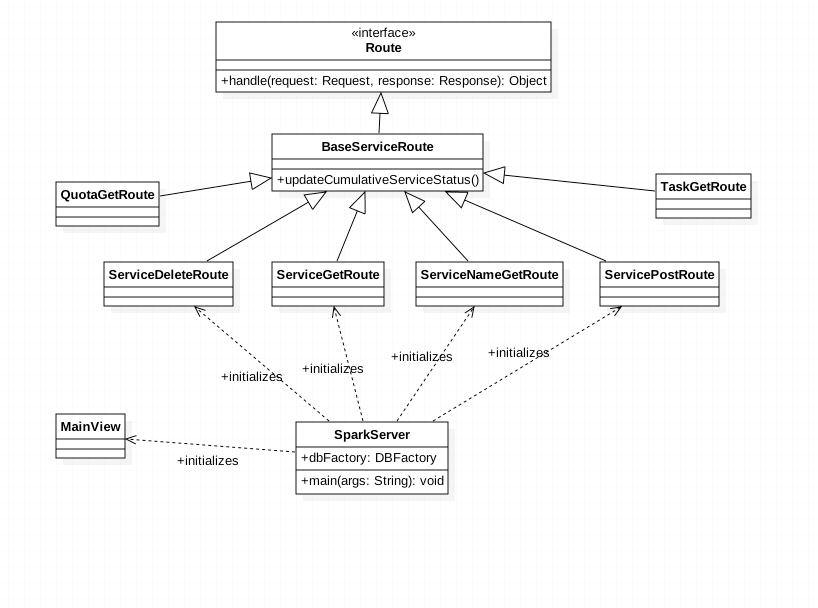


Figure - OpenStack Server Routes

The above diagram shows the SparkServer – which is the embedded Jetty Server that is started when the application bootstraps. It initializes the views in the system and routes (which serve as REST endpoints).

* 1. Base Service Route – is abstract class with common methods that are used in some of the Route implementations.
  2. Service Delete Route – maps to DELETE /virtapp/api/v1.0/services.
  3. Service Get Route – maps to GET /virtapp/api/v1.0/services
  4. Service Name Get Route – maps to GET /virtapp/api/v1.0/services/:serviceId
  5. Service Post Route – maps to POST /virtapp/api/v1.0/services
  6. Task Get Route – maps to GET /virtapp/api/v1.0/tasks
  7. Quota Get Route – maps to GET /virtapp/api/v1.0/quota
  8. Main View – is the renderer for single page application Web UI and maps to /openstack/index.ftl (where, ftl stands for freemarker template).

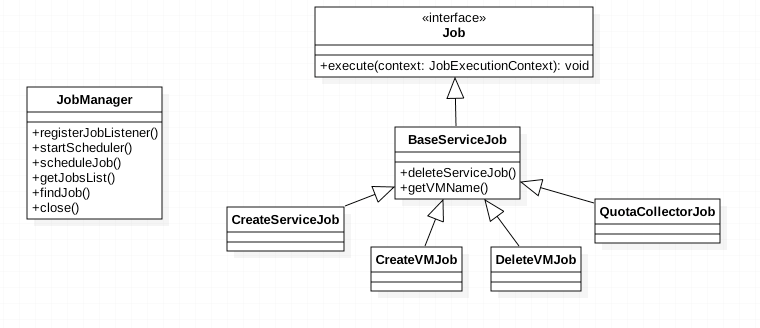


Figure – Asynchronous Job Classes

Lastly, the above class diagram depicts the job classes in the system.

* 1. **Create Service Job** – job to create a service asynchronously. The service creation once triggered from the endpoint (ServicePostRoute) returns immediately to the user with the message service creation has been scheduled.
  2. **Delete VM Job** – job to delete a service.
  3. **Create VM Job** – is a child job of create service job and is triggered per VM so several VMs can be created in parallel.
  4. **Quota Collector Job** – uses SSH to collect resource usage by a tenant/user combination. Following is the SSH command currently being used:

**source ~/keystonerc\_admin; nova-manage service describe\_resource --host={0}**

and

**source ~/keystonerc\_admin; nova usage-list --start {0} --end {1}**

# Implementation

## Overview

The **OpenStack cloud self-service portal** will have the following components:

1. **OpenStack client layer** - which will interface with the OpenStack service using the REST APIs provided by OpenStack modules (like Nova API, Neutron API and Glance API). We will use [OpenStack4J](http://www.openstack4j.com/) library which provides Java object for the OpenStack entities.
2. **MongoDB database access layer** - which will interface with the MongoDB NoSQL database to persist the portal user credentials and session information.
3. **Self Service Portal REST API** - will provide REST interface and will be hosted on the Jetty embedded web server (using [Spark java](http://sparkjava.com/) framework).
4. **Web UI** - will be dynamic HTML responsive UI that renders on both small and large devices (PCs and mobile) properly. For this we will use CSS [bootstrap](http://getbootstrap.com/) library by Twitter. The UI will be a single page application built using [Freemarker Template Engine](http://www.freemarker.incubator.apache.org/).

## Web UI

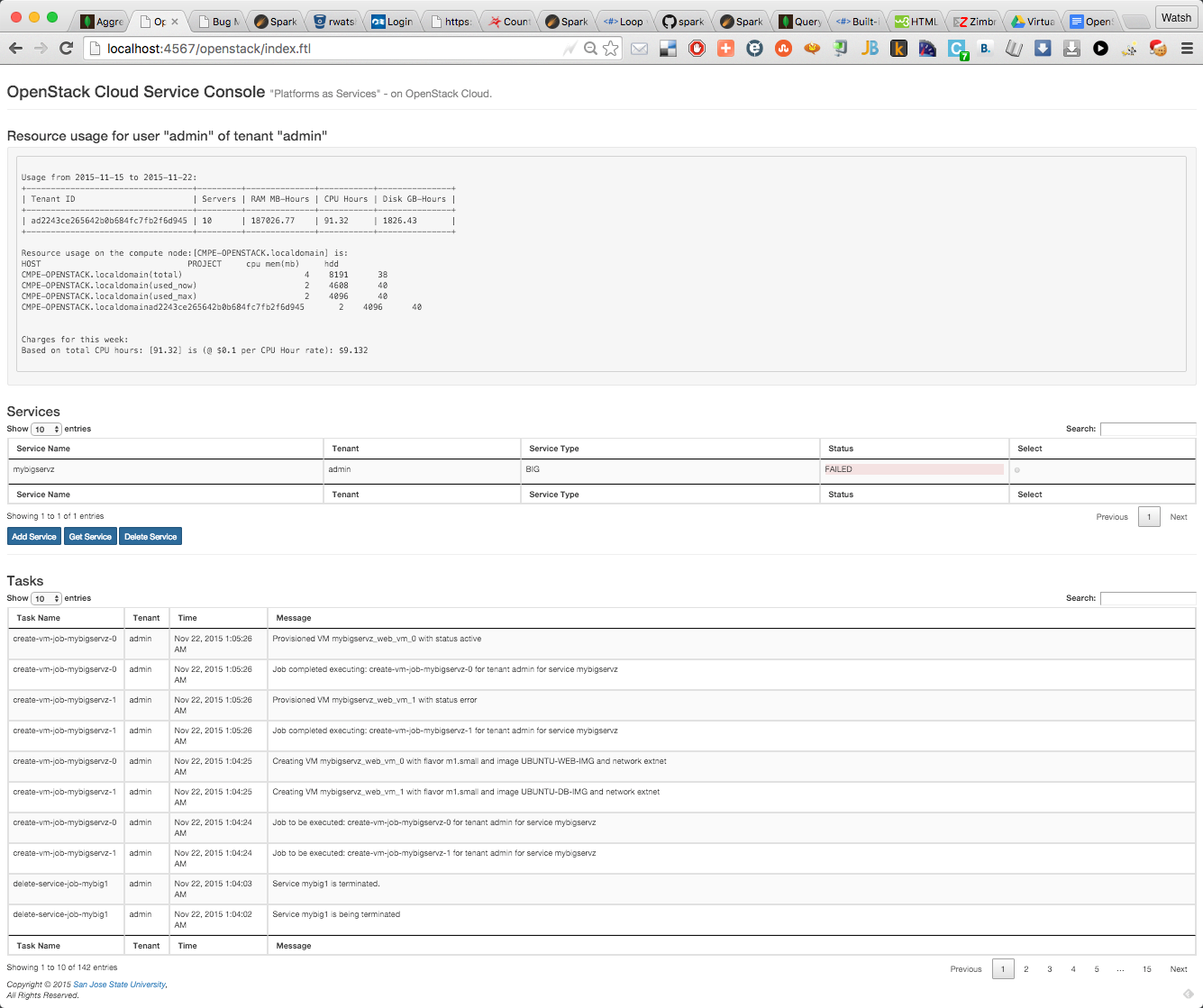


Figure – A Single page application (Web UI)

There are 3 sections in the Web UI:

* 1. Resource usage – shows the usage for the last week and the computed CPU hour rate based on the configurable $0.1 per CPU hour value.
  2. Services – shows a list of services and provides interfaces to create and delete services.
  3. Task – shows the list of events in the system.

The UI tables can be sorted, paginated and searched.

## Resource Usage

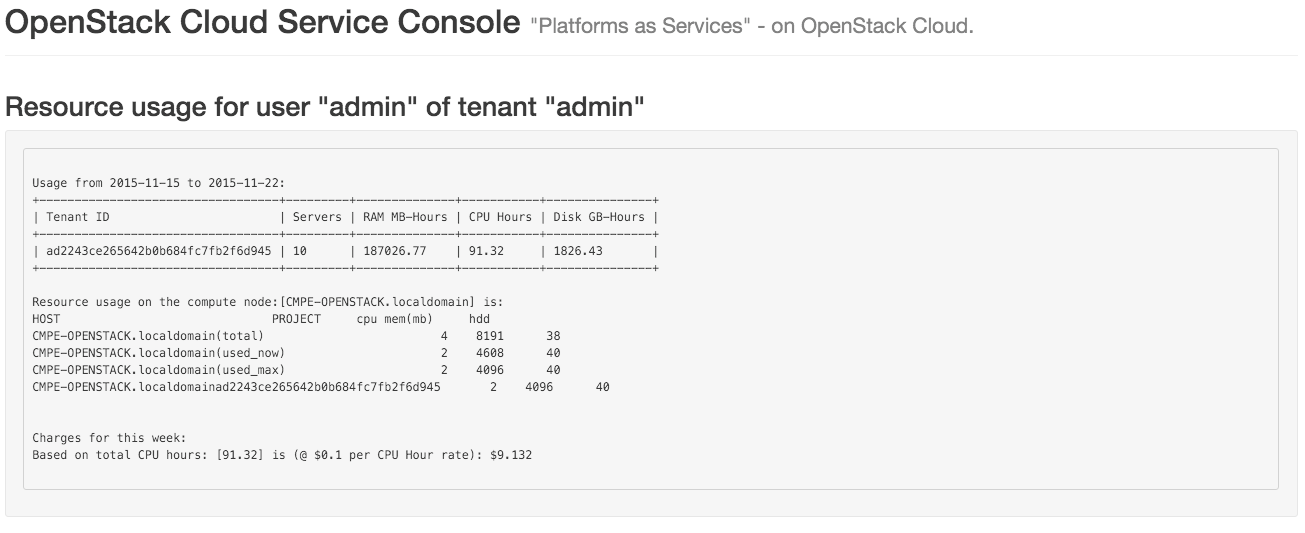


Figure - Resource Usage

The first usage metrics comes from the following command:

**nova usage-list --start {0} --end {1}**

The start and end dates are from 1 week ago to current date.

This table shows the number of servers used by the tenant over the past week, memory hours, CPU hours and disk GB hours.

The second usage metrics is collected on a per compute node basis. For our project we have just one compute node so the data is just for one node as given by the following command:

**nova-manage service describe\_resource --host={0}**

This command provides data on total, max and current CPU, memory and HDD usage. This data is presented but not used in calculating the charge back for tenant currently.

Lastly, we show the charges for the week based on the CPU hours metrics from the first usage metrics. The rate per CPU hour is defaulted to $0.1 and is configurable in the system (requires server to be restarted).

## Services

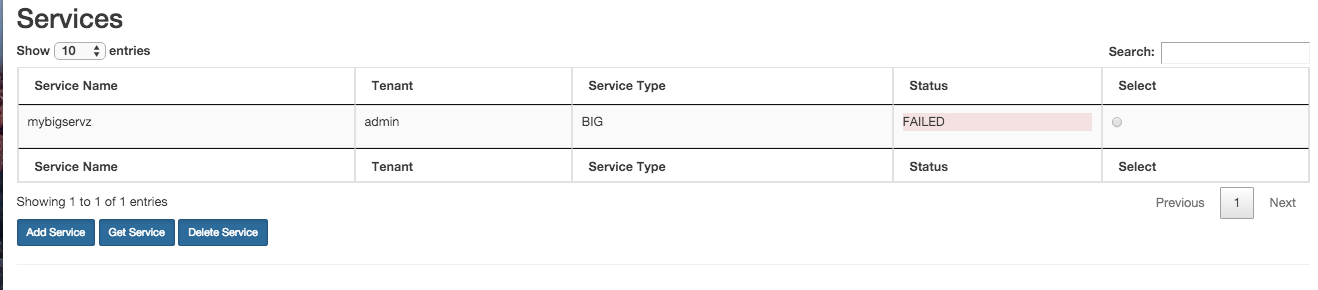


Figure - Services Table

The above Services table shows the list of services created by the tenant user.

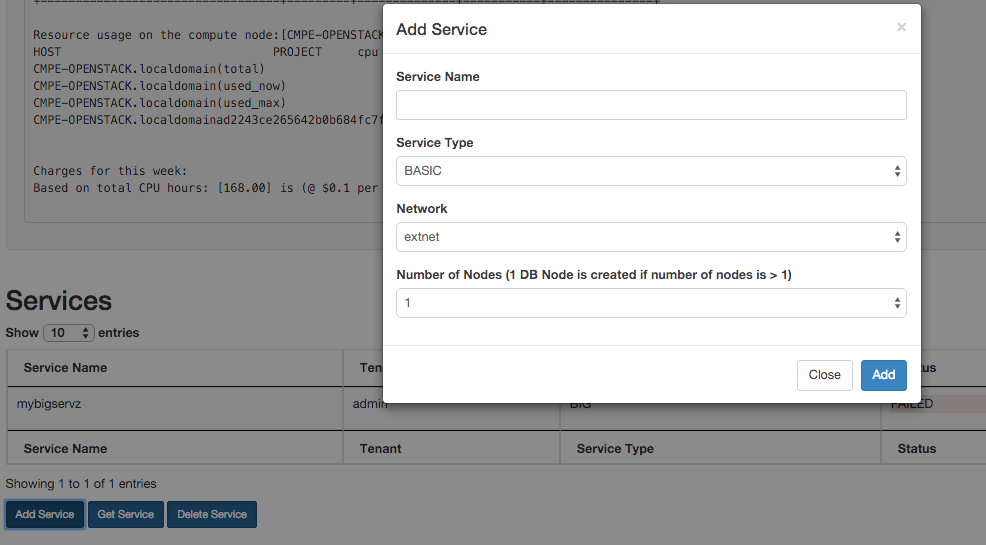


Figure - Add Service

The Add Service dialog shows the following fields:

* 1. Service Name – A name for the service. This should be unique for a given tenant.
  2. Service Type – Choice between BASIC and BIG
  3. Network – choice of the list of networks available on the server
  4. Number of Nodes – 1 DB node is created if number of nodes specified is > 1. This field is only enabled for BIG service type.

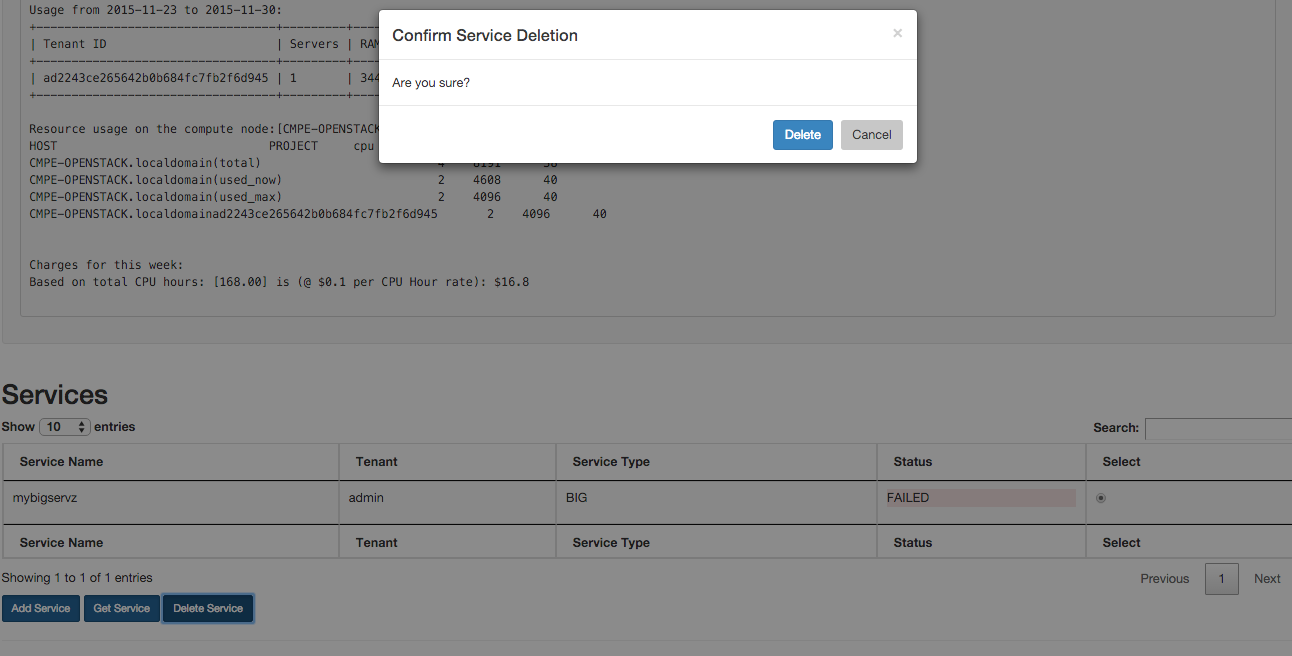


Figure - Delete Service

When Delete service button is clicked for the selected service then the delete confirmation dialog is shown.

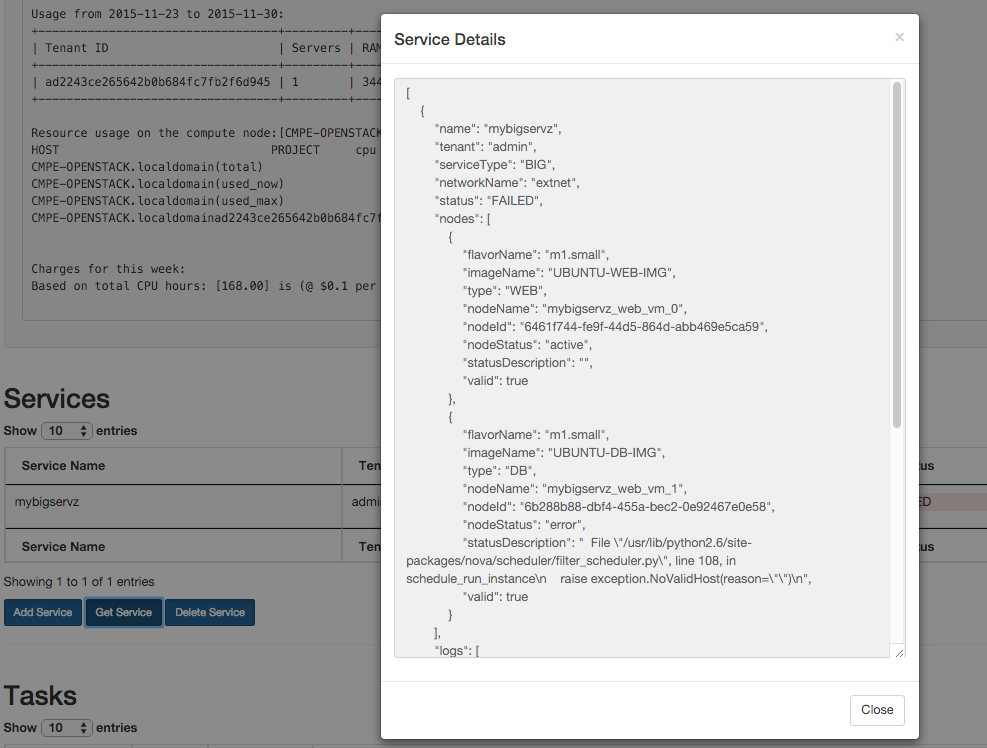


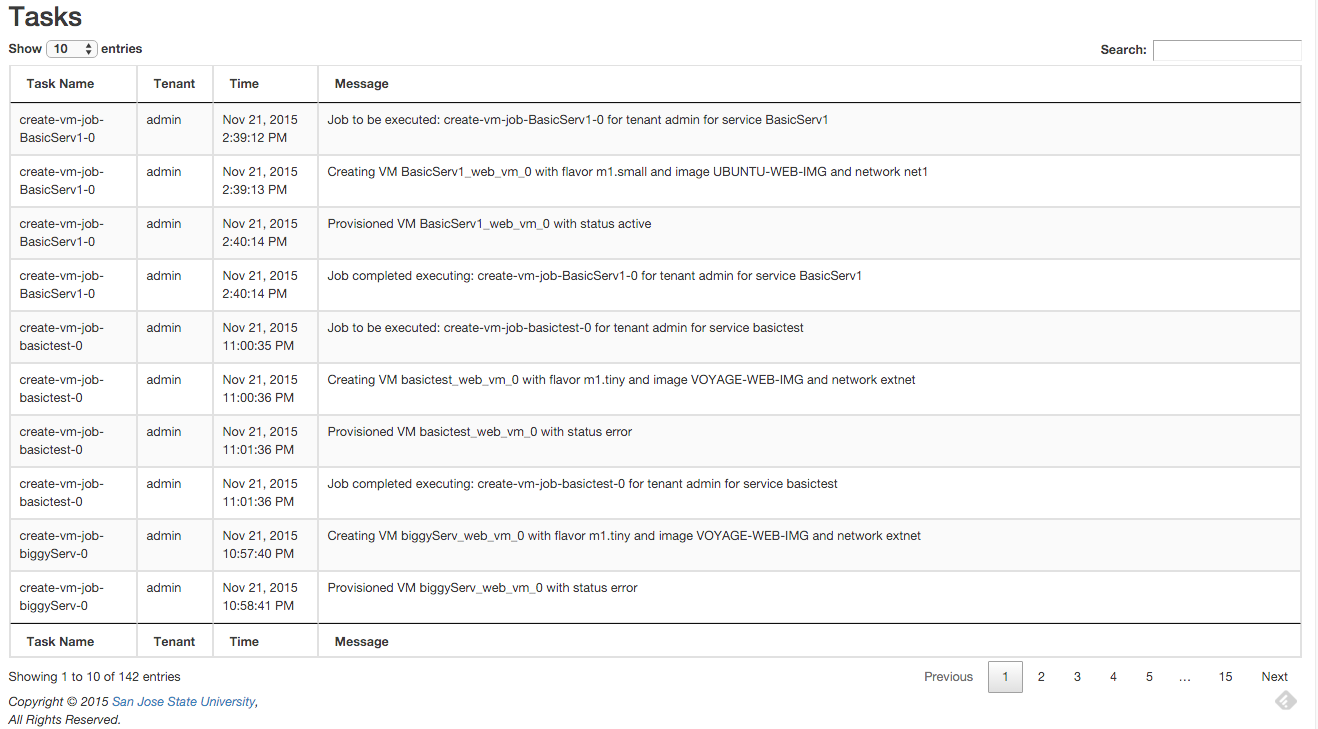
Figure - Get Service

Not all details for the service is shown in the table. To get the entire service details user can click on the Get Service button after selecting the service in the table above. This will fetch a detailed JSON message showing all information maintained for the service by the system. Most importantly it shows the VMs that are part of the service, their name, image, flavor, status etc. If the VM provisioning failed then it will also the error in OpenStack that led to the failure in provisioning the VM.

Below is the dump of the entire Service JSON:

|  |
| --- |
| [  {  "name": "mybigservz",  "tenant": "admin",  "serviceType": "BIG",  "networkName": "extnet",  "status": "FAILED",  "nodes": [  {  "flavorName": "m1.small",  "imageName": "UBUNTU-WEB-IMG",  "type": "WEB",  "nodeName": "mybigservz\_web\_vm\_0",  "nodeId": "6461f744-fe9f-44d5-864d-abb469e5ca59",  "nodeStatus": "active",  "statusDescription": "",  "valid": true  },  {  "flavorName": "m1.small",  "imageName": "UBUNTU-DB-IMG",  "type": "DB",  "nodeName": "mybigservz\_web\_vm\_1",  "nodeId": "6b288b88-dbf4-455a-bec2-0e92467e0e58",  "nodeStatus": "error",  "statusDescription": " File \"/usr/lib/python2.6/site-packages/nova/scheduler/filter\_scheduler.py\", line 108, in schedule\_run\_instance\n raise exception.NoValidHost(reason=\"\")\n",  "valid": true  }  ],  "logs": [  {  "message": "Job to be executed: create-vm-job-mybigservz-0 for tenant admin for service mybigservz",  "time": 1448183064321  },  {  "message": "Job to be executed: create-vm-job-mybigservz-1 for tenant admin for service mybigservz",  "time": 1448183064325  },  {  "message": "Job completed executing: create-vm-job-mybigservz-0 for tenant admin for service mybigservz",  "time": 1448183126672  }  ],  "valid": true  }  ] |

## Tasks



The above UI shows the tasks in the system. Tasks represent the events in the system as the scheduled asynchronous jobs execute in the background. Since the job of creating, deleting a service happen in the background so we maintain the logs for every event of interest for the user to inspect to be informed of the progress in the job execution.

# Observations and Challenges

1. **No console access for VMs created via OpenStack -** The host system for VirtualBox may support the hardware virtualization (like Intel VT-x) feature and that gets used by VirtualBox itself. The VMs inside VirtualBox may provide further virtualization (as in our case the OpenStack VM used Xen hypervisor to provide virtualization) but the hardware virtualization feature will not be available to it. For console to work, the VMs require the hardware virtualization feature to be available for its immediate hypervisor. Due to this limitation we will only be able to install an OS without a desktop environment.
2. **OpenStack deployment in all-in-one VM mode** made the resources scarce for provisioning VMs especially in the case of provisioning a BIG service which has 2 or more VMs.
3. **External network access for VMs provisioned on OpenStack** did not work even after setting up a router between VM network and external network.
4. **Needed to host the application on the OpenStack VM itself** asthe OpenStack endpoint URLs for the various REST services is maintained in an endpoints table in OpenStack’s MySQL DB. So if trying to access the OpenStack keystone REST endpoint goes through then other endpoints returned from the DB have the local IP address of the OpenStack VM host and it is not accessible from outside the OpenStack VM.
5. **Python –** We played with

# Authors

Project group 3 members:

1. Sagar Anil Mangalani
2. Mehak Pavagadhi
3. Nolyda Tep
4. Nikita Mathur
5. Nilay Kothari
6. Sachet Hegde
7. Watsh Rajneesh

Following is how the work was divided in the team:

1. UI Design – Mehak and Nikita
2. REST API for services – Nolyda and Nilay
3. Mongo DB integration – Watsh and Sachet
4. OpenStack client - Watsh
5. UI integration with backend - Sagar
6. Testing – All for their own module.
7. Project Report and Slides – All contributed.

# Bibliography

Following material (mostly online text and lecture slides) were used to understand and implement the project:

* 1. [OpenStack Course Slides](https://sjsu.instructure.com/courses/1171302/files/folder/Project) on Canvas.
  2. [OpenStack user manual](http://docs.openstack.org/user-guide/index.html) for understanding the architecture OpenStack.
  3. [OpenStack4J user manual](http://openstack4j.com/learn/getting-started/).
  4. [Mongo DB user manual](https://docs.mongodb.org/manual/) – for understanding the NoSQL DB.
  5. [Spark Java](http://sparkjava.com/) Micro Web Application Framework.
  6. [Quartz Scheduler](http://www.quartz-scheduler.org/documentation/2.2.1/pdf/index) user manual – for implementing the background processing of service request.
  7. [Freemarker Template Engine user manual](http://freemarker.incubator.apache.org/docs/index.html) – for implementing the presentation layer of the Web UI.