Alarm Counter

A Ceilometer – OpenStack Application

Tejas Tovinkere Pattabhi
UTD VOLUNTEER AT AWARD SOLUTIONS Summer 2015

Contents

1 Introduction

The Alarm Counter Application is designed to work in OpenStack environment to do the following:

1. Receive data from Web hooks created by Ceilometer Alarms, reporting the status of each alarm machine.
2. Portray the above information graphically on a web page.

These tasks are accomplished by creating a web service application – a set of Web pages to host the content and a Server Application which reads, aggregates data from Web Hooks and feed the Web Pages when requested.

The following technologies are used to build the application:

- The Server Component: Python
- The Client Component: HTML5, CSS3, JavaScript, JQuery
- Web Hooks: OpenStack – Ceilometer

2 Pre-Requisites

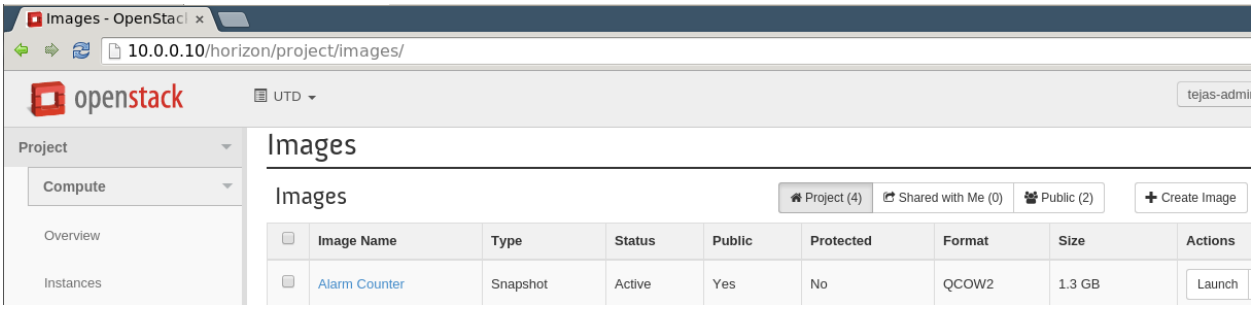
The engineer working on this Application should have minimum knowledge on the following to build/modify/update this application.

- Git
- HTML, JQuery, JavaScript
- Python
- Working in Ubuntu Environment
- IP Addressing and Sub-netting (For floating IPs)
- OpenStack – Ceilometer
- Apache2 Web Service
- TCP/IP Conceptual Programming

2.1 Server Creation

The simplest way for server creation in OpenStack is to create an instance of the Public Image ‘Alarm Counter’. This will create an instance with the above desired code, running the server application on port 5623 in the background by default.

The instance can be seen in the images:



If there are some issues seen in creating this instance, you can setup the server manually.

2.2 Manual Setup of Application/Web Server

Step 1: Download the image of Ubuntu Server for Cloud. The OpenStack website suggests the following link: http://uec-images.ubuntu.com/trusty/current/trusty-server-cloudimg-amd64-disk1.img and create an image of it on OpenStack. Chose format as 'RAW' and NOT 'IMG'

Step 2: Create an instance on a tenant of your choice, network of your choice. Make sure whatever network you connect, its router is gateway-ed to EXT-NET. Without which nothing would work. **Important:** Use the following script in the post-creation script section. If you fail using this, you will not be able to login to the machine you create.

```
#cloud-config
password: mySuperSecret!
chpasswd: { expire: False }
ssh_pwauth: True
```

Step 3: Launch the instance. Open the Console through OpenStack, wait for it to boot. On booting, login with the following credentials:

```
Username: ubuntu
Password: mySuperSecret!
```

Note: This password is limited to this instance only. If you take a snapshot of this instance, the username and password will erase.

To overcome the above limitation, create a user of your own and change the password of root. Use the following commands:

```
$ sudo passwd root
// Enter the password for user Ubuntu first.
```

```
// New Password for root
// Confirm New Password for root

$ sudo adduser <custom_username>
// Fill out the password and other personal information

$ sudo adduser <custom_sername> sudo
// This is to add the user created to sudo group.
```

Now, if you take a snapshot, the above user and root password will be retained.

Step 4: Check if the machine is able to connect to the internet. Perform the following actions:

```
$ ping 8.8.8.8
```

If there is a positive response, then you now have to add the DNS server. If there is issues with Internet connectivity, refer the Appendix of this document to troubleshoot internet connectivity.

To add the DNS server, you need to login as a sudo user. Perform the following actions:

```
$ cd /etc/resolvconf/resolv.conf.d/
$ sudo vi base
```

Enter the following contents:

```
nameserver 8.8.8.8
nameserver 8.8.4.4
```

Save the contents and exit

Now you need to restart the network service, in order to reflect the changes made to DNS configurations.

```
$ sudo service network-manager restart
```

Now, if you try to ping http://www.google.com/, it should work. If not refer the Appendix for troubleshooting.

Note: What you do here, will be recorded permanently and will not erase on rebooting the machine.

Step 5: Install the following packages with the associated commands

Package: Git

```
$ sudo apt-get install git
```

Package: Apache2

```
$ sudo apt-get install apache2
```

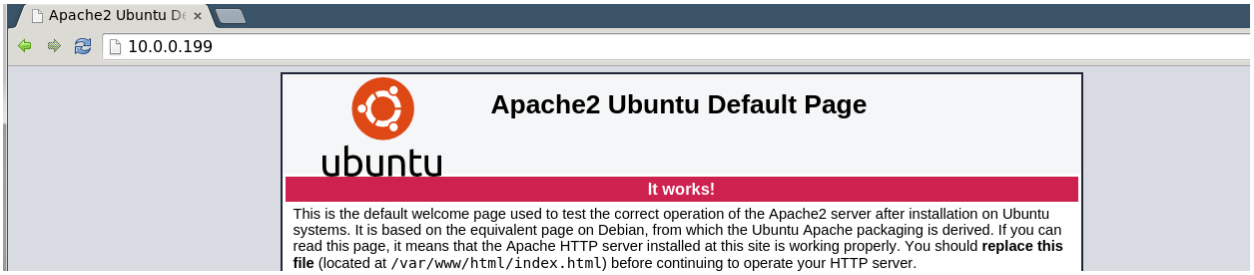
Package: lamp-server

```
$ sudo apt-get install lamp-server^
```

Step 6: Associate your current Machine with a Floating IP of EXT-NET. You can do that by choosing the option on horizon.

Record the IP address associated. This will be needed in future. **This document will refer to the Floating IP as 10.0.0.199.**

Step 7: Check if your Apache2 is installed accurately, and if floating IP associated is working right. To do this Open your browser on login server and type your floating IP on the address bar. You should see a default Apache2 web page like this.



Step 8: If you have succeeded with no issues so far, now you can make this server run your application and web pages. For which you need to clone a repository. Follow these commands on a terminal logged in with a sudo user.

```
$ cd /var/www/html/
$ sudo git clone https://tejaspattabhi@bitbucket.org/tejaspattabhi/alarm-counter.git
```

This will clone the entire repository and creates a folder called ‘alarm-counter’. This folder hosts both the web pages and the server component of the application.

Step 9: You need to run your server component. You can either run it manually by doing so:

```
$ cd /var/www/html/alarm-counter
$ sudo python server.py
```

Or you can make it run as a daemon process in the background. For which you’ll have to do the following:

```
$ cd /etc/init.d
$ sudo vi startservice.sh
```

Enter the following contents in it:

```
#!/bin/sh
sudo python /var/www/html/alarm-counter/server.py
return 0;
```

Note: Do not miss the return 0;

Now you need to add this script to rc.d (run command by default)

```
$ sudo chmod 0755 startservice.sh
$ sudo update-rc.d startservice.sh defaults
$ sudo update-rc.d startservice.sh enable
```

Now, reboot your server. The 'server.py' is running at background by default.

Step 10: Last step, you need to update the code for the IP address in three locations

- Alarm-counter/js/custom.js
- Alarm-counter/js/overview.js
- Alarm-counter/js/flush.js

Locate the following code: $.ajax

```
$.ajax({
        'url': 'http://10.0.0.199:5623/?request=data',
```

Update the IP address to your current floating IP. DO NOT CHANGE THE PORT.

Now, the server is up and ready!

You can take a snapshot of it for future reference.

3 Application Usage

The application developed can be used for Monitoring every Machine for which the Ceilometer alarm has been configured. To create a ceilometer alarm, please refer the create alarm section of this chapter.

3.1 Overview

There are three pages:

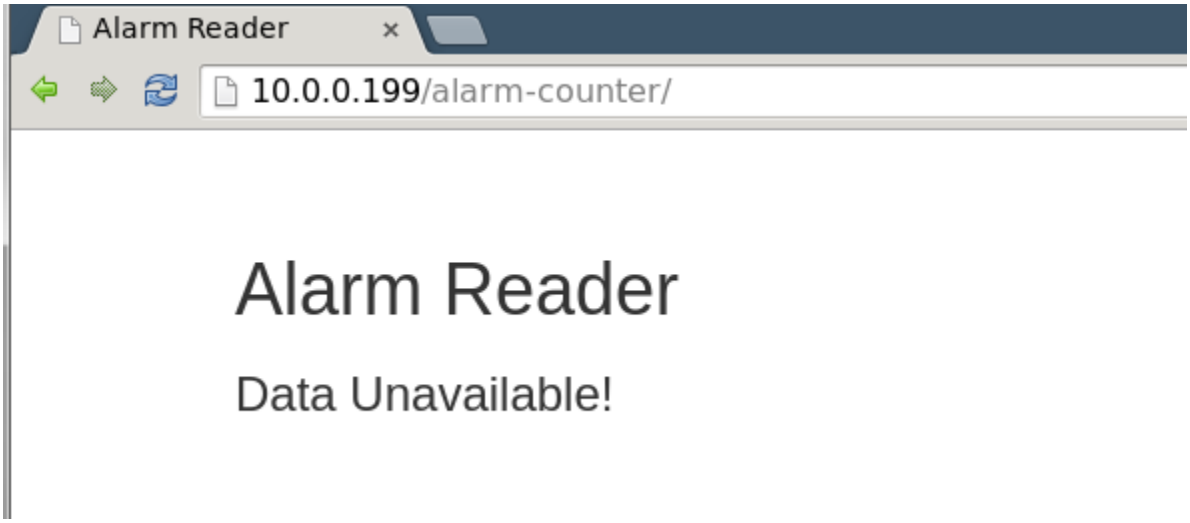
alarm-counter/index.html – Overview of all tenants having VMs with Ceilometer Alarms

alarm-counter/detailed.html – Detail view of every VM in every tenant for its current state and history of alarms.

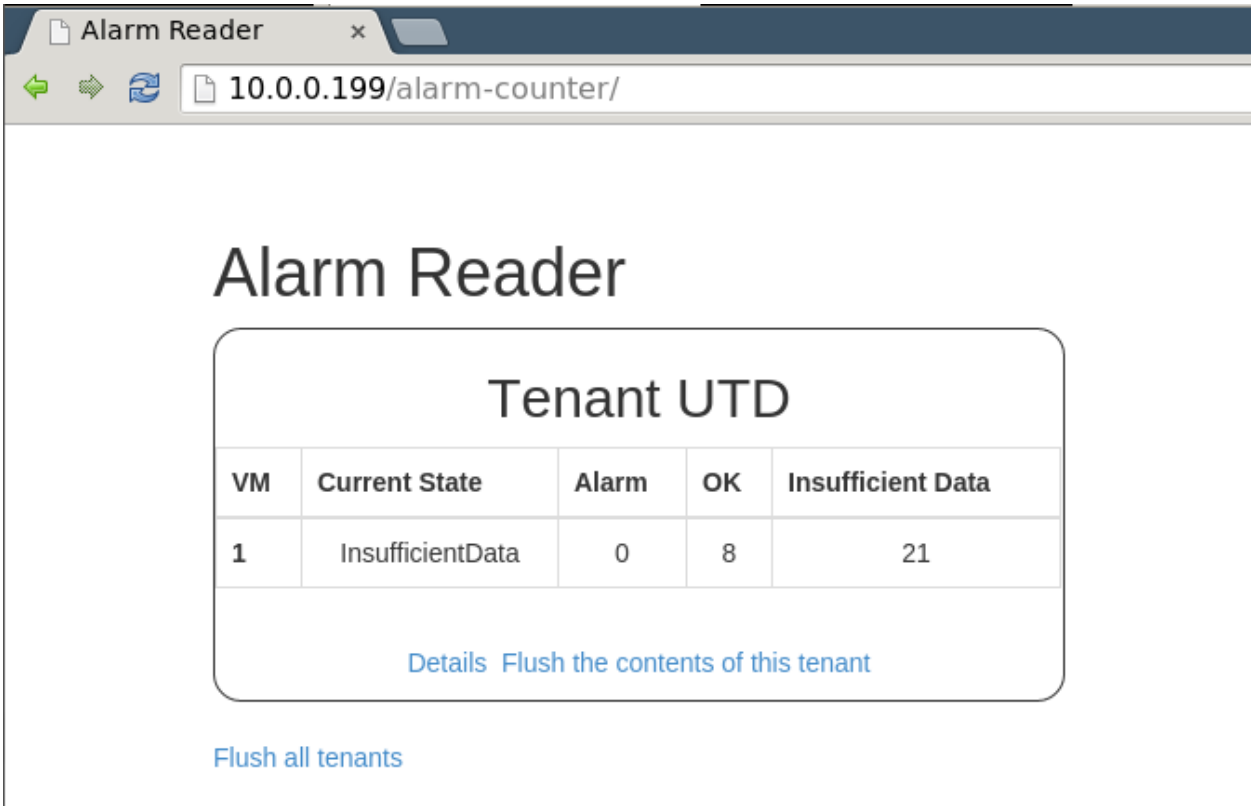
Alarm-counter/flush.html – Clears data for a tenant or all tenants. This is a passive page. It has to be redirected from the above two pages.

You can find the following screenshots

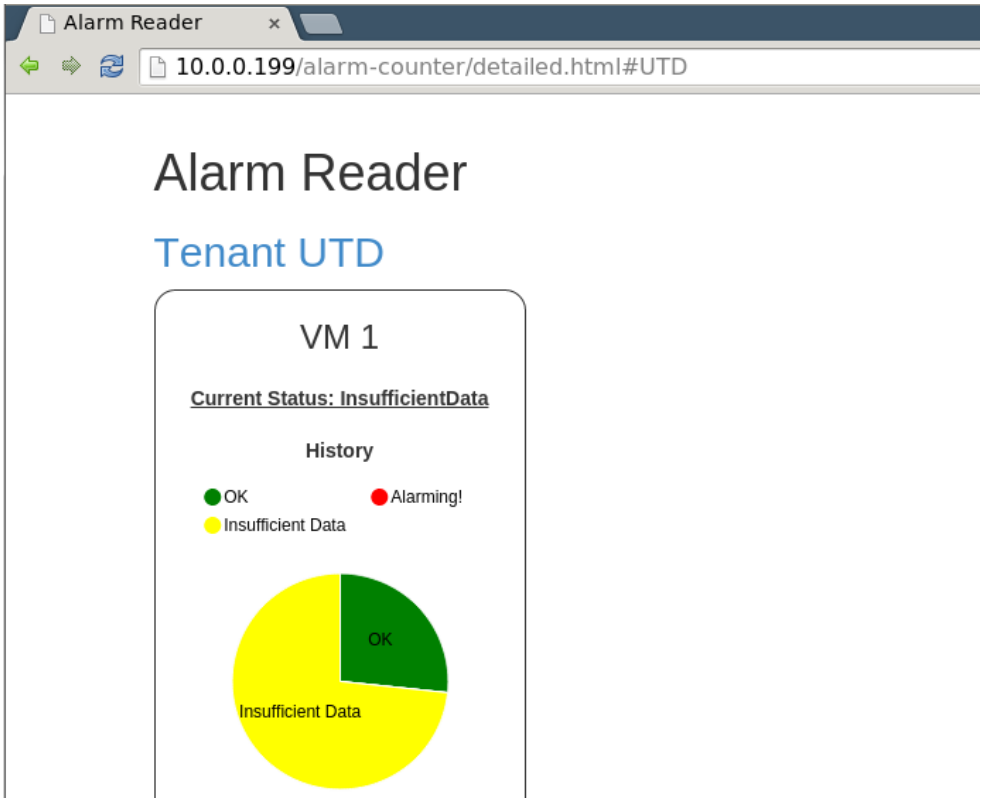
If there is no data you can see something like this:



If you have a VM with Ceilometer alarm configured, you can see something like this:



You can click on ‘Details’ anchor, you will see the following:



You can click on ‘Flush all’ or ‘Flush this tenant’ for appropriate actions.

3.2 Alarm Creation

For this, first you need to create a VM in a tenant. Note down the ID of that VM. It can be seen in the Overview tab on the horizon -> instances -> VM

Then use the following commands to create the alarm.

Note: These commands have to execute on a terminal on login server.

```
export OS_TENANT_NAME=<your Tenant in which VM has been created>
export OS_USERNAME=<OpenStack Login ID>
export OS_PASSWORD=<OpenStack Login Password>
export OS_AUTH_URL="http://10.0.0.10:5000/v2.0/"
export OS_SERVICE_ENDPOINT="http://10.0.0.10:35357/v2.0"
export OS_SERVICE_TOKEN=password

ceilometer alarm-threshold-create --name VM-1 \
                                  --description 'instance running hot' \
                                  --meter-name cpu_util  \
                                  --threshold 7.45 \
                                  --comparison-operator gt \
                                  --statistic avg \
                                  --period 20 \
```

```
  --evaluation-periods 3 \
   --alarm-action
'http://10.0.0.199:5623/?tenant=UTD&alarm=Alarming&VMID=1' \
   --insufficient-data-action
'http://10.0.0.199:5623/?tenant=UTD&alarm=InsufficientData&
VMID=1' \
   --ok-action
'http://10.0.0.199:5623/?tenant=UTD&alarm=OK&VMID=1' \
  --repeat-actions True \
  --query resource_id=<VM ID noted from horizon>
```

The following should be noted:

- The URLs seen on alarm-action, insufficient-data-action and ok-actions are the web hooks.
- Please note that tenant name, alarm name and VMID should be accurately mentioned here. If not, it will aggregate inaccurately.
- VMID is just a Virtual ID – Integer what you assign to your VM for easy interpretation.
- If any of the above fields are missing, the content will be rejected by the server.
- The IP address should be your Floating IP of the server. Please do not change the PORT.

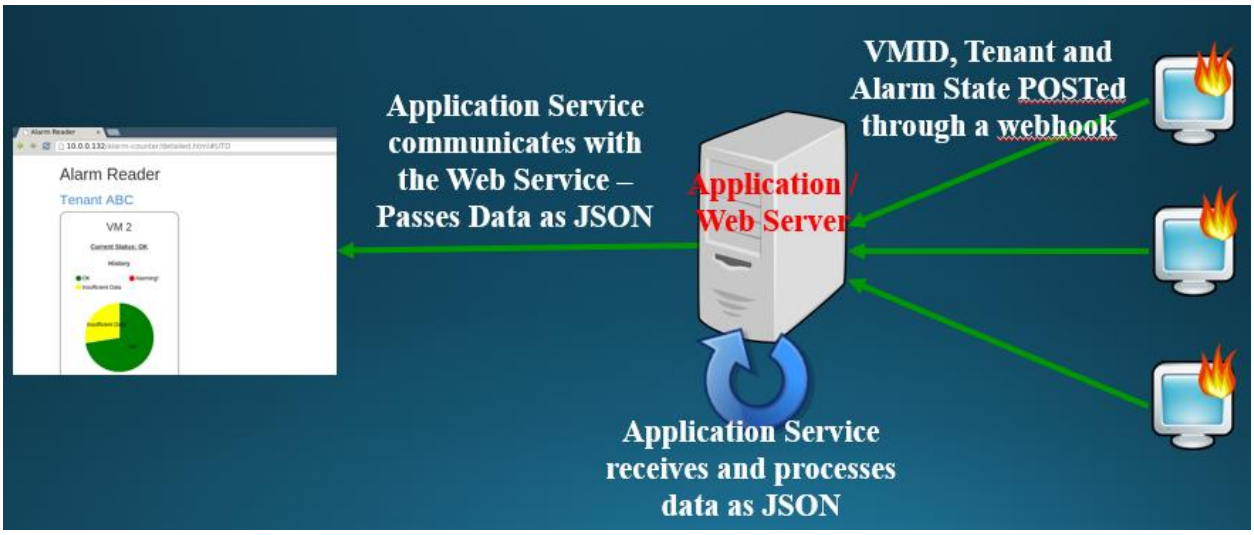
Please refer the OpenStack Manuals for detailed explanation.

4 Server Application

The Server is built using Python HTTP Services. A base HTTP server is setup on the port 5623 (That's why you can see the :5623 in the code and URLs!). 5623 is chosen as a port in random. The idea is to not oppose any standard ports. It comprises of only one file – server.py from the repository. It does not require any parameters to run.

4.1 Architecture

The server is built to follow this architecture:



It mainly serves two kind of requests:

4.1.1 POST request coming from the web hooks

The web hook URLs are executed every fixed period of time (provided *repeat_actions* are set *True* in Ceilometer Alarm). This period is again set in the Ceilometer Alarm. These are the requests coming as POST from the Ceilometer alarms. They tend to POST some information. In our case, it is the set of parameters coming along with the URL.

A standard URL looks like this:

http://10.0.0.199:5623/?tenant=X&VMID=3&alarm=OK

Here,

- 10.0.0.199 – is the floating IP.
- 5623 – is the port
- Tenant=X – X is the name of the tenant where the VM is created.
- VMID=ID – ID is an integer/string which the user has assigned to the VM he is using.
- Alarm=OK – Is the current state of the alarm. This can take 3 possible values. *OK*, *Alarm* and *Insufficient Data.*

This information is grabbed and stored in memory. Content of each VM in each tenant is aggregated. It is sent as JSON upon requested.

4.1.2 GET request coming from web pages

As seen in the architecture, the web pages make a GET request to the server application. These request URLs can be of 3 kinds:

http://10.0.0.199:5623/?request=data

This URL is sent when the client needs the data aggregated by the server. Upon receiving such a request, the server throws all the contents that it has stored and aggregated till the point of request as JSON with appropriate headers.

http://10.0.0.199:5623/?flush=all

This URL is sent when the client needs to flush all data that it has aggregated till the point of request being made. The response would be a simple 200 success header following with a HTML content saying “Success”.

http://10.0.0.199:5623/?flush=TENANT_NAME

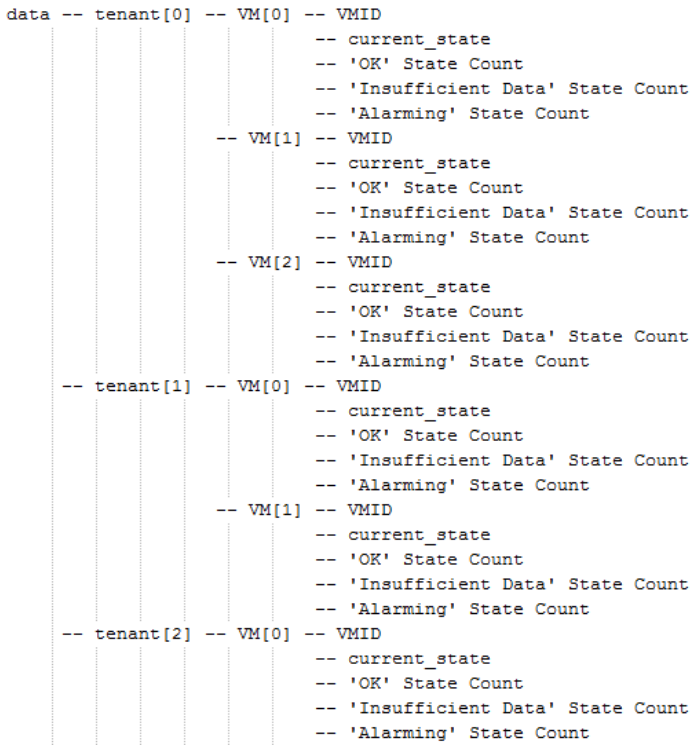
This URL is sent when the client needs to flush a particular Tenant’s data that it has aggregated till the point of request being made. The response would be a simple 200 success header following with a HTML content saying “Success” if the Tenant Name is valid. If it’s not, it just returns a 200 with an error message “No Tenant found with the name as specified”.

4.2 Working of Server, Aggregation and Data Structure

Every time data is collected from a web hook, the server stores it in a Python Dictionary. For an end user, it can be seen as a simple Hash Map [Redundant]. Redundant - Each Value in the hash map will have another hash map (dictionary).

4.3 Data Structure

This way, the VMs can be grouped based on Tenants (as many VMs can exist in one tenant and there can be many tenants). The following diagram gives a visualization of how the data is stored.



As the picture indicates, *data* is a dictionary (key, value), where key is the name of the tenant and value is another dictionary. So there can be any number of distinct tenants which can be referenced in this structure. Each value of this dictionary is another dictionary, where key is the VMID of the VM in that tenant and value is a dictionary with the following elements:

- current_state: Indicates the current state of the VM
- 'OK' State Count: Indicates the number of times the VM has reported OK in the past.
- 'Insufficient Data State' Count: Indicates the number of times the VM has reported Insufficient Data in the past.
- 'Alarming' State Count: Indicates the number of times the VM has reported Alarm in the past.

This way, many VMs with distinct VMIDs can be maintained under a single tenant.

4.4 Working

Every time a POST request occurs, the URL will be of type mentioned in Section 4.1.1 . The server follows these steps:

Step 1: Parse the parameters for VMID, tenant name and alarm state.

Step 2: Check if the tenant exists in *data*. If not, initialize an empty dictionary in the name of the tenant. Go to step 4.

Step 3: If yes, check if the VMID exists in data[tenant]. If present, update the current state and increment the alarm state count by 1.

Step 4: If not, initialize the empty dictionary and add the current state to be the reported state. The count of the current state to be 1 and the rest to 0.

Every time a GET request occurs, the URL will be of type mentioned in Section 4.1.2. The server follows these steps:

Step 1: Parse the parameters for VMID, tenant name and alarm state.

Step 2: Check if 'request=data' is present in the parameters. If yes, dump the contents of *data* to JSON and send the HTTP response back as 200 with appropriate headers.

Step 3: If not, check if 'flush' is present. Then if flush=all, set *data* to empty valued dictionary (clearing all data). Or else if flush=tenant_name, delete the contents of the data[tenant] if present. And return 200 with appropriate headers.

This way, the server application interfaces between the web pages and the Ceilometer alarm – web hooks.

5 Client Application

Client Application is a composition of web pages designed using HTML5, CSS3, JavaScript and JQuery. The CSS3 used is the Bootstrap framework. The most important point to note here is the graphical representation of data is done with the help of "Data Driven Documents", in other words "D3.js".

All of these frameworks have been made local to the environment. None of these would require an external reference.

The client application comprises of 3 major web pages. Please refer the Overview section of this document for further details.

6 Appendix

6.1 Network Troubleshooting

For the application to work, you need the following to be working:

- PING – ICMP protocol
- Web Service TCP/IP port 80 HTTP
- Web Service TCP/IP port 443 HTTPS
- SSH Service TCP/IP port 22

For all the above, you need to create a security group and open all the above ports in both INGRESS and EGRESS modes. Then you need to create your server instance in this security group. There is a default group which accomplishes all these. There is also a OPEN group which opens all ports. You can use these or create your own.

Note: Make sure your machine is somehow routed to EXT-NET. Without which none of the following will work.

After using a group with the above options, test the following:

6.1.1 Test 1: Ping

Check if

```
$ ping 8.8.8.8
```

Works. If not, check the MTU by:

```
$ ifconfig
```

If eth0 ifconfig is set around 1500, then use the following command

```
$ sudo ifconfig eth0 mtu 1454
```

Then recheck the ping command. It should work.

Next, let's check the DNS:

```
$ ping http://www.google.com/
```

If this does not work, follow Step 2 of server creation.

6.1.2 Test 2: HTTP/HTTPS

```
$ curl -k http://www.google.com/
```

If you get a response, you are all set.

If not, check for MTU as suggested in Test 1: ping

6.1.3 Test 3: SSH

On your login server, try the following:

```
$ ssh <custom_username>@<Floating IP>
```

You should be able to connect with your correct password, if not, got back and check your MTU on console.