C12G LABS

Public Cloud Computing with OpenNebula 1.4

C12G Labs S.L.

Rev20100611

C12G LABS

Contents

1 Getting Started **5**
- 1.1 Building a Public Cloud . . . 5
  - 1.1.1 What is a Public Cloud? . . . 5
  - 1.1.2 The User View . . . 5
  - 1.1.3 How the System Operates . . . 6

2 OGF OCCI API **7**
- 2.1 Configuration Guide . . . 7
  - 2.1.1 Overview . . . 7
  - 2.1.2 Requirements & Installation . . . 7
  - 2.1.3 Configuration . . . 8
  - 2.1.4 Starting the Cloud Service . . . 11
  - 2.1.5 Cloud Users . . . 12
- 2.2 Usage Guide . . . 12
  - 2.2.1 OCCI Resources . . . 12
  - 2.2.2 User Account Configuration . . . 12
  - 2.2.3 Hello OCCI Cloud! . . . 13

3 EC2-Query API subset **15**
- 3.1 Configuration Guide . . . 15
  - 3.1.1 Overview . . . 15
  - 3.1.2 Requirements & Installation . . . 15
  - 3.1.3 Configuration . . . 16
  - 3.1.4 Starting the Cloud Service . . . 19
  - 3.1.5 Cloud Users . . . 19
- 3.2 Usage Guide . . . 19
  - 3.2.1 User Account Configuration . . . 20
  - 3.2.2 Hello Cloud! . . . 20

Chapter 1

Getting Started

1.1 Building a Public Cloud

1.1.1 What is a Public Cloud?

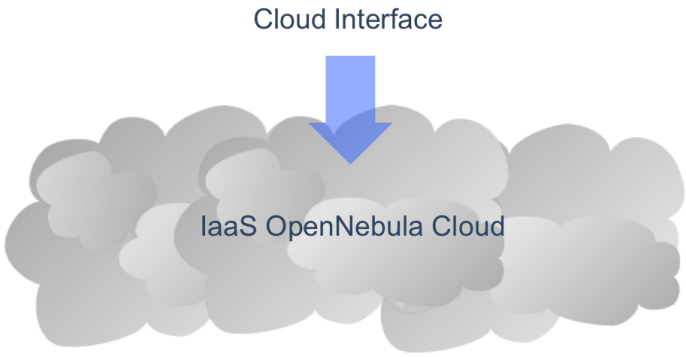

Figure 1.1:

A Public Cloud is an **extension of a Private Cloud to expose RESTful Cloud interfaces**. Cloud interfaces can be added to your Private or Hybrid Cloud if you want to provide partners or external users with access to your infrastructure, or to sell your overcapacity. Obviously, a local cloud solution is the natural back-end for any public cloud.

1.1.2 The User View

The following interfaces provide a **simple and remote management of cloud (virtual) resources at a high abstraction level**:

- EC2 Query subset
- RESERVOIR Cloud Interface and OGF OCCI (planned for 1.4.2)

Users will be able to use commands that **clone the functionality of the EC2 Cloud service**. Starting with a working installation of an OS[1] residing on an **.img** file, with three simple steps a user can launch it in the cloud.

First, they will be able to **upload** it to the cloud using:

After the image is uploaded in OpenNebula repository, it needs to be **registered** to be used in the cloud:

Now the user can **launch** the registered image to be run in the cloud:

Additionally, the instance can be **monitored** with:

[1] Operating System

1.1.3 How the System Operates

There is **no modification in the operation of OpenNebula to expose Cloud interfaces**. Users can interface the infrastructure using any Private or Public Cloud interface.

C12G LABS

Chapter 2

OGF OCCI API

2.1 Configuration Guide

2.1.1 Overview

The OpenNebula OCCI is a web service that enables you to launch and manage virtual machines in your OpenNebula installation using the latest draft of the OGF OCCI API specification. The OpenNebula OCCI service is implemented upon the new **OpenNebula Cloud API**[1] (OCA) layer that exposes the full capabilities of an OpenNebula private cloud; and Sinatra, a widely used light web framework.

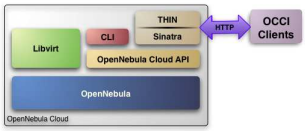

Figure 2.1:

The current implementation includes all the resource referenced in the latest draft of the OGF OCCI API[1] specification, namely:

- **Compute**: launching, retrieve, update and deletion
- **Network**: creation, retrieve and deletion
- **Storage**: upload, retrieve and deletion

The following sections explain how to install and configure the OCCI service on top of a running OpenNebula cloud.

The OpenNebula OCCI service provides an OCCI interface to your cloud instance, that can be used alongside the native OpenNebula CLI, the libvirt interface or even the EC2 Query API[1]

The OpenNebula distribution includes the tools needed to use the OpenNebula OCCI service

2.1.2 Requirements & Installation

You must have an OpenNebula site properly configured and running to install the OpenNebula OCCI service, be sure to check the OpenNebula Installation and Configuration Guides to set up your private cloud first. This guide also assumes that you are familiar with the configuration and use of OpenNebula.

The OpenNebula OCCI service was installed during the OpenNebula installation, so you just need to install the following packages to meet the runtime dependencies:

- The Sinatra web framework and the thin web server:

[1] Application Programming Interface

- The libraries for the Storage Repository and Client Tools:

`curb` is just neccessary to upload files faster. If not installed, upload file with `occi-storage -M` that uses the `multipart-post` library

2.1.3 Configuration

The service is configured through the `$ONE_LOCATION/etc/occi-server.conf` file, where you can set up the basic operational parameters for the OCCI service, namely:

- **Administration Account**, the web server need to perform some operation using the `oneadmin` account, mainly to check the identity of the cloud users. You have to specify the `USER` and `PASSWORD` of `oneadmin`.
- **Connection Parameters**, the xml-rpc service of the `oned` daemon; and the server and port for the OpenNebula OCCI service web server. This will be the URL[2] of your cloud.
- **Storage Repository**, the storage repository provides an easy-to-use repository to store Compute images. You need to specify the `DATABASE` and `IMAGE_DIR` for this service.
- **Computes**, the name of the bridge that the VM needs to connect to in the physical host to get network connection. See the Managing Virtual Networks guide for more details.
- **Filesystem format**, the default format in which the empty filesystems (needed for certain Computes) will be formatted in. In case of ommission, by default is ext3.
- **Compute Types**, a `VM_TYPE` defines the name and the OpenNebula templates for each type of Compute, to enable different *sizes* of Computes.

The following table summarizes the available options:

VARIABLE	VALUE
USER	name for the oneadmin account
PASSWORD	oneadmin password
ONE_XMLRPC	oned xmlrpc service, http://localhost:2633/RPC2
SERVER	FQDN for your cloud
PORT	for incoming connections
DATABASE	for the Storage repository
IMAGE_DIR	to store *cloud* images
BRIDGE	Name of the bridge needed to create Networks
FS_FORMAT	to store *cloud* images
VM_TYPE	The Computes types for your cloud

The `SERVER` **must** be a FQDN, do not use IP's here
`IMAGE_DIR` **must** be an existing directory
Preserve bash syntax in the `occi-server.conf` file
Example:

```
# OpenNebula administrator user
USER=oneadmin
PASSWORD=mypass

# OpenNebula sever contact information
ONE_XMLRPC=http://localhost:2633/RPC2

# Host and port where OCCI service will run
SERVER=cloud.opennebula.org
```

[2] Uniform Resource Locator

```
PORT=4567

# Configuration for the image repository
DATABASE=/srv/cloud/one/var/occi.db
IMAGE_DIR=/srv/cloud/images/

# Configuration for OpenNebula's Virtual Networks
BRIDGE=eth0

# Default format for FS
FS_FORMAT=ext3

# VM types allowed and its template file (inside templates directory)
VM_TYPE=[NAME=small,  TEMPLATE=small.erb]
VM_TYPE=[NAME=medium, TEMPLATE=medium.erb]
VM_TYPE=[NAME=large,  TEMPLATE=large.erb]
```

Configuring a SSL proxy

OpenNebula OCCI runs natively just on normal HTTP[3] connections. If the extra security provided by SSL[4] is needed, a proxy can be set up to handle the SSL[4] connection that forwards the petition to the OCCI Service and takes back the answer to the client.

This set up needs:

- A server certificate for the SSL[4] connections
- An HTTP[3] proxy that understands SSL[4]
- OCCI Service configuration to accept petitions from the proxy

If you want to try out the SSL[4] setup easily, you can find in the following lines an example to set a self-signed certificate to be used by a lighttpd configured to act as an HTTP[3] proxy to a correctly configured OCCI Service.

Let's assume the server were the lighttpd proxy is going to be started is called `cloudserver.org`. Therefore, the steps are:

1. Snakeoil server certificate We are going to generate a snakeoil certificate. If using an Ubuntu system follow the next steps (otherwise your milleage may vary, but not a lot):

- Install the `ssl-cert` package
- Generate the certificate
- As we are using lighttpd, we need to append the private key with the certificate to obtain a server certificate valid to lighttpd

2. lighttpd as a SSL HTTP proxy You will need to edit the `/etc/lighttpd/lighttpd.conf` configuration file and

- Add the following modules (if not present already)
  - mod_access
  - mod_alias
  - mod_proxy
  - mod_accesslog

[3] Hyper Text Transfer Protocol
[4] Secure Sockets Layer

  - mod_compress

- Change the server port to 443 if you are going to run lighttpd as root, or any number above 1024 otherwise:

```
server.port               = 8443
```

- Add the proxy module section:

```
#### proxy module
## read proxy.txt for more info
proxy.server              = ( "" =>
                              ("" =>
                               (
                                 "host" => "127.0.0.1",
                                 "port" => 4567
                               )
                               )
                            )


#### SSL engine
ssl.engine                = "enable"
ssl.pemfile               = "/etc/lighttpd/server.pem"
```

The host must be the server hostname of the computer running the EC2Query Service, and the port the one that the EC2Query Service is running on.

3.OCCI Service configuration The `econe.conf` needs to define the following:

```
# Host and port where OCA server will run
SERVER=127.0.0.1
PORT=4567
# SSL proxy that serves the API (set if is being used)
SSL_SERVER=cloudserver.org
```

Once the lighttpd server is started, OCCI petitions using HTTPS uris can be directed to `https://cloudserver.org:8443`, that will then be unencrypted, passed to localhost, port 4567, satisfied (hopefully), encrypted again and then passed back to the client.

Defining Compute types

You can define as many Compute types as you want, just:

- Create a template for the new type and place it in `$ONE_LOCATION/etc/occi_templates`. This template will be *completed* with the data for each cloud *occi-vm create* request, and then submitted to OpenNebula. You can start by modifying the `small.erb` example, to adjust it to your cloud:

```
NAME   = <%= @vm_info['NAME']%>

CPU    = 1
MEMORY = 1024

OS = [ kernel     = /vmlinuz,
       initrd     = /initrd.img,
```

```
       root       = sda1,
       kernel_cmd = "ro xencons=tty console=tty1"]

<% if vm_info['STORAGE']
       vm_info['STORAGE'].each do |key, image|
  image=[image].flatten
case key
    when "SWAP"
        image.each do |img|
%>
DISK = [ type = "swap",
         size=<%= img['size']%>,
         target=<%= img['dev']%> ]
<%
        end
    when "DISK"
        image.each do |img|
%>
DISK = [ type = "disk",
         target=<%= img['dev']%>,
         source=<%= img['source']%>,
         image_id=<%= img['image']%> ]
<%
        end
    when "FS"
        image.each do |img|
%>
DISK = [ type = "fs",
         target=<%= img['dev']%>,
         size=<%= img['size']%>,
         format=<%= @config[:fs_format]||"ext3"%>  ]
<%      end %>
<% end %>
<% end %>
<% end %>
<% if vm_info['NETWORK'] and vm_info['NETWORK']['NIC'] %>
<% vm_info['NETWORK']['NIC'].each do |nic| %>
NIC = [
<% if nic['ip'] %>
        IP=<%= nic['ip'] %>,
<% end %>
        NETWORK="<%= nic['network']%>",
        NETWORK_ID=<%= nic['network_id'] %>
]
<% end %>
<% end %>
INSTANCE_TYPE = <%= vm_info[:instance_type ]%>
```

The templates are processed by the OCCI service to include specific data for the instance, you should not need to modify the <%= ... %> compounds. Start by adjusting the OS[5], CPU and MEMORY to your needs

2.1.4 Starting the Cloud Service

To start the OCCI service just issue the following command

[5] Operating System

You can find the econe server log file in `$ONE_LOCATION/var/occi-server.log` if OpenNebula has been installed in standalone, or in `/var/log/one/occi-server.log` if installed in system-wide.

To stop the OCCI service:

2.1.5 Cloud Users

The cloud users have to be created in the OpenNebula system by `oneadmin` using the `oneuser` utility. Once a user is registered in the system, using the same procedure as to create private cloud users, they can start using the system. The users will authenticate using the HTTP basic authentication with `user-ID` their OpenNebula's username and `password` their OpenNebula's password.

2.2 Usage Guide

The OpenNebula OCCI API[1] is a RESTful service to create, control and monitor cloud resources based on the latest draft of the OGF OCCI API specification. Interactions with the resources are done through HTTP[3] verbs (**GET**, **POST**, **PUT** and **DELETE**).

2.2.1 OCCI Resources

There are three kind of resources, listed below with their implemented methods:

- **Storage**:
  - Upload: using a multi-part HTTP[3] POST.
  - Retrieve: using a HTTP[3] GET.
- **Network**:
  - Upload: using a HTTP[3] POST.
  - Retrieve: using a HTTP[3] GET.
- **Compute**:
  - Upload: using a HTTP[3] POST.
  - Update: using a HTTP[3] PUT.
  - Retrieve: using a HTTP[3] GET.

Collections of resources have the following implemented methods for all Pool Resources are:

- **POST**: Creates a new resource within the colletion, with the representation in the POST body.
- **GET**: Retrieves the representation of the collection.

2.2.2 User Account Configuration

An account is needed in order to use the OpenNebula OCCI cloud. The cloud administrator will be responsible for assigning these accounts, which have a one to one correspondence with OpenNebula accounts, so all the cloud administrator has to do is check the configuration guide to setup accounts, and automatically the OpenNebula OCCI cloud account will be created.

In order to use such an account, the end user can make use of clients programmed to access the services described in the previous section. For this, she has to set up her environment, particularly the following aspects:

- **Authentication**: This can be achieved in two different ways, here listed in order of priority (i.e. values specified in the argument line supersede environmental variables)
  - Using the **commands arguments**. All the commands accept a **username** (as the OpenNebula username) and a **password** (as the OpenNebula password)

– If the above is not available, the **ONE_AUTH** variable will be checked for authentication (with the same used for OpenNebula CLI, pointing to a file containing a single line: "username:password").

- **Server location**: The command need to know where the OpenNebula OCCI service is running. You can pass the OCCI service endpoint using the "url" flag in the commands. If that is not present, the **OCCI_URL**[2] environment variable is used (in the form of a http URL[2], including the port if it is not the standard 80). Again, if the **OCCI_URL**[2] variable is not present, it will default to "http://localhost:4567"

The **OCCI_URL**[2] has to use the FQDN of the OCCI Service

2.2.3 Hello OCCI Cloud!

Lets take a walk through a typical usage scenario. In this brief scenario it will be shown how to upload an image to the OCCI OpenNebula Storage repository, how to create a Network in the OpenNebula OCCI cloud and how to create Compute resource using the image and the network previously created.

- **Storage**

Assuming we have a working Gentoo installation residing in an **.img** file, we can upload it into the OpenNebula OCCI cloud using the following OCCI representation of the image:

```
<DISK>
    <NAME>GentooImage</NAME>
    <URL>file:////images/gentoo.img</URL>
</DISK>
```

Next, using the **occi-storage** command we will create the Storage resource:

$ ./occi-storage –url http://localhost:4567 –username oneadmin –password opennebula create imagexml <DISK><ID>ab5c9770-7ade-012c-f1d5-00254bd6f386</ID><NAME>GentooImage</NAME><SIZE>1000</

The user should take note of this **ID**, as it will be needed to add it to the Compute resource.

- **Network**

The next step would be to create a Network resource

```
<NETWORK>
  <NAME>MyServiceNetwork</NAME>
  <ADDRESS>192.168.1.1</ADDRESS>
  <SIZE>200</SIZE>
</NETWORK>
```

Next, using the **occi-network** command we will create the Network resource:

$ ./occi-network –url http://localhost:4567 –username oneadmin –password opennebula create vnxml <NIC><ID>23</ID><NAME>MyServiceNetwork</NAME><ADDRESS>192.168.1.1</ADDRESS><SIZE>200</

- **Compute**

The last step would be to create a Compute resource referencing the Storage and Networks resource previously created by means of their ID, using a representation like the following:

```
<COMPUTE>
        <NAME>MyCompute</NAME>
        <STORAGE>
                <SWAP size="1024" dev="sda2"/>
                <DISK image="ab5c9770-7ade-012c-f1d5-00254bd6f386" dev="sda1"/>
                <FS size="512" format="ext3" dev="sda3"/>
        </STORAGE>
        <NETWORK>
                <NIC network="23" ip="192.168.0.9"/>
        </NETWORK>
        <INSTANCE_TYPE>small</INSTANCE_TYPE>
</COMPUTE>
```

Next, using the **occi-compute** command we will create the Compute resource:

$ ./occi-compute –url http://localhost:4567 –username tinova –password opennebula create ~/vmxml
<COMPUTE><ID>17</ID><NAME>MyCompute</NAME><STATE>PENDING</STATE><STORAGE><DIS
image=”ab5c9770-7ade-012c-f1d5-00254bd6f386” dev=”sda1”/><FS size=”512” format=”ext3” dev=”sda3”/><SWAP
size=”1024” dev=”sda2”/></STORAGE><NETWORK><NIC network=”23” ip=”192.168.0.9”/><NIC
network=”23” ip=”192.168.1.1”/></NETWORK></COMPUTE>

You can obtain more information on how to use the above commands accessing their Usage help passing them the **-h** flag

In platforms where 'curl' is not available or buggy (i.e. CentOS), a '-M' option is available to perform upload using the native ruby Net::HTTP[3] using http multipart

C12G LABS

Chapter 3

EC2-Query API subset

3.1 Configuration Guide

3.1.1 Overview

The OpenNebula EC2 Query is a web service that enables you to launch and manage virtual machines in your OpenNebula installation through the Amazon EC2 Query Interface. In this way, you can use any EC2 Query tool or utility to access your Private Cloud. The EC2 Query web service is implemented upon the new **OpenNebula Cloud API**[1] (OCA) layer that exposes the full capabilities of an OpenNebula private cloud; and Sinatra, a widely used light web framework.

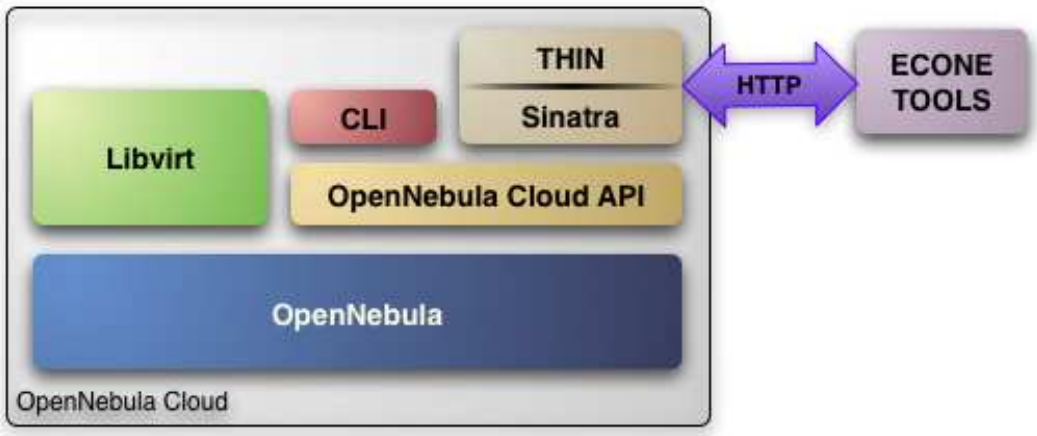

Figure 3.1:

The current implementation includes the basic routines to use a Cloud, namely: image upload and registration, and the VM run, describe and terminate operations. The following sections explains you how to install and configure the EC2 Query web service on top of a running OpenNebula cloud.

The OpenNebula EC2 Query service provides a Amazon EC2 Query API[1] compatible interface to your cloud, that can be used alongside the native OpenNebula CLI or the libvirt interface.

The OpenNebula distribution includes the tools needed to use the EC2 Query service.

3.1.2 Requirements & Installation

You must have an OpenNebula site properly configured and running to install the EC2 Query service, be sure to check the OpenNebula Installation and Configuration Guides to set up your private cloud first. This guide also assumes that you are familiar with the configuration and use of OpenNebula.

[1] Application Programming Interface

The EC2 Query service was installed during the OpenNebula installation, so you just need to install the following packages to meet the runtime dependencies:

- The Amazon EC2 Query API[1] library:
- The Sinatra web framework and the thin web server:
- The libraries for the Image Repository and Client Tools (packages names are taken from the Ubuntu distribution):

3.1.3 Configuration

The service is configured through the `$ONE_LOCATION/etc/econe.conf` file, where you can set up the basic operational parameters for the EC2 Query web service, namely:

- **Administration Account**, the web server need to perform some operation using the `oneadmin` account, mainly to check the identity of the cloud users. You have to specify the `USER` and `PASSWORD` of `oneadmin`.
- **Connection Parameters**, the xml-rpc service of the `oned` daemon; and the server and port for the EC2_URL[2]. This will be the URL[2] of your cloud.
- **Image Repository**, the image repository provides a easy-to-use and simple replacement of the S3 service to store and upload images. You need to specify the `DATABASE` and `IMAGE_DIR` for this service.
- **Virtual Machine Types**, a `VM_TYPE` defines the name and the OpenNebula templates for each type.

The following table summarizes the available options:

VARIABLE	VALUE
USER	name for the oneadmin account
PASSWORD	oneadmin password
ONE_XMLRPC	oned xmlrpc service, http://localhost:2633/RPC2
SERVER	FQDN for your cloud
PORT	for incoming connections
DATABASE	for the Image repository
IMAGE_DIR	to store *cloud* images
VM_TYPE	The VM types for your cloud

The `SERVER` **must** be a FQDN, do not use IP's here.
`IMAGE_DIR` **must** be an existing directory
Preserve bash syntax in the `econe.conf` file
Example:

```
# OpenNebula administrator user
USER=oneadmin
PASSWORD=mypass

# OpenNebula sever contact information
ONE_XMLRPC=http://localhost:2633/RPC2

# Host and port where OCA server will run
SERVER=cloud.opennebula.org
PORT=4567
# SSL proxy that serves the API (set if is being used)
```

[2]Uniform Resource Locator

```
#SSL_SERVER=fqdn.of.the.server


# Configuration for the image repository
DATABASE=/srv/cloud/one/var/ec2.db
IMAGE_DIR=/srv/cloud/images/

# VM types allowed and its template file (inside templates directory)
VM_TYPE=[NAME=m1.small, TEMPLATE=m1.small.erb]
VM_TYPE=[NAME=m1.medium, TEMPLATE=m1.medium.erb]
```

Configuring a SSL proxy

OpenNebula EC2 Query Service runs natively just on normal HTTP[3] connections. If the extra security provided by SSL[4] is needed, a proxy can be set up to handle the SSL[4] connection that forwards the petition to the EC2 Query Service and takes back the answer to the client.

This set up needs:

- A server certificate for the SSL[4] connections
- An HTTP[3] proxy that understands SSL[4]
- EC2Query Service configuration to accept petitions from the proxy

If you want to try out the SSL[4] setup easily, you can find in the following lines an example to set a self-signed certificate to be used by a lighttpd configured to act as an HTTP[3] proxy to a correctly configured EC2 Query Service.

Let's assume the server were the lighttpd proxy is going to be started is called `cloudserver.org`. Therefore, the steps are:

1. Snakeoil server certificate We are going to generate a snakeoil certificate. If using an Ubuntu system follow the next steps (otherwise your milleage may vary, but not a lot):

- Install the `ssl-cert` package
- Generate the certificate
- As we are using lighttpd, we need to append the private key with the certificate to obtain a server certificate valid to lighttpd

2. lighttpd as a SSL HTTP proxy You will need to edit the `/etc/lighttpd/lighttpd.conf` configuration file and

- Add the following modules (if not present already)
  - mod_access
  - mod_alias
  - mod_proxy
  - mod_accesslog
  - mod_compress
- Change the server port to 443 if you are going to run lighttpd as root, or any number above 1024 otherwise:

[3] Hyper Text Transfer Protocol
[4] Secure Sockets Layer

```
server.port               = 8443
```

- Add the proxy module section:

```
#### proxy module
## read proxy.txt for more info
proxy.server              = ( "" =>
                              ("" =>
                               (
                                 "host" => "127.0.0.1",
                                 "port" => 4567
                               )
                               )
                            )


#### SSL engine
ssl.engine                = "enable"
ssl.pemfile               = "/etc/lighttpd/server.pem"
```

The host must be the server hostname of the computer running the EC2Query Service, and the port the one that the EC2Query Service is running on.

3. EC2Query Service configuration The `econe.conf` needs to define the following:

```
# Host and port where OCA server will run
SERVER=127.0.0.1
PORT=4567
# SSL proxy that serves the API (set if is being used)
SSL_SERVER=cloudserver.org
```

Once the lighttpd server is started, EC2Query petitions using HTTPS uris can be directed to `https://cloudserver.org:8` that will then be unencrypted, passed to localhost, port 4567, satisfied (hopefully), encrypted again and then passed back to the client.

Defining VM types

You can define as many Virtual Machine types as you want, just:

- Create a template for the new type and place it in `$ONE_LOCATION/etc/ec2query_templates`. This template will be *completed* with the data for each cloud *run-instance* request, and then submitted to OpenNebula. You can start by modifying the `m1.small.erb` example, to adjust it to your cloud:

```
NAME   = eco-vm

CPU    = 1
MEMORY = 1024

OS = [ kernel     = /vmlinuz,
       initrd     = /initrd.img,
       root       = sda1,
       kernel_cmd = "ro xencons=tty console=tty1"]

DISK = [ source   = <%= @vm_info[:img_path] %>,
         clone    = no,
```

```
        target   = sda1,
        readonly = no]

NIC = [ network = "Public EC2" ]

IMAGE_ID = <%= @vm_info[:img_id] %>
INSTANCE_TYPE = <%= @vm_info[:instance_type ]%>
```

- Add a `VM_TYPE` attribute to `$ONE_LOCATION/etc/eco.conf` with the NAME for the new type and the `TEMPLATE` that should be use:

```
VM_TYPE=[NAME=m1.large, TEMPLATE=m1.large.erb]
```

The templates are processed by the EC2 server to include specific data for the instance, you should not need to modify the <%= ... %> compounds. Start by adjusting the OS[5], CPU and MEMORY to your needs.

Networking for the Cloud VMs

By default, the templates used to instantiate the virtual machines includes a `NIC` interface to be attached to a virtual network named `Public EC2`. You **have to create this network** using the `onevnet utility` with the IP's you want to lease to the VMs created with the EC2 Query service.

3.1.4 Starting the Cloud Service

To start the EC2 Query service just issue the following command

You can find the econe server log file in `$ONE_LOCATION/var/econe-server.log` if OpenNebula has been installed in standalone, or in `/var/log/one/econe-server.log` if installed in system-wide.

To stop the EC2 Query service:

3.1.5 Cloud Users

The cloud users have to be created in the OpenNebula system by `oneadmin` using the `oneuser` utility. Once a user is registered in the system, using the same procedure as to create private cloud users, they can start using the system. The users will authenticate using the Amazon EC2 procedure with `AWSAccessKeyId` their OpenNebula's username and `AWSSecretAccessKey` their OpenNebula's password.

3.2 Usage Guide

The EC2 Query API offers the functionality exposed by Amazon EC2: upload images, register them, run, monitor and terminate instances, etc. In short, Query requests are HTTP[3] or HTTPS requests that use the HTTP[3] verb GET or POST and a Query parameter.

OpenNebula implements a subset of the EC2 Query interface, enabling the creation of public clouds managed by OpenNebula. In this first release of the API[1] implementation, the methods implemented are:

- **upload image**: Uploads an image to the repository manager
- **register image**: Registers an image (previously uploaded in the repository manager) in order to be launched, check this link for the method description.
- **describe images**: Lists all registered images belonging to one particular user.

* **run instances**: Runs an instance of a particular image (that needs to be referenced), check this link for the method d

[5] Operating System

```
*
**[[http://docs.amazonwebservices.com/AWSEC2/2009-04-04/APIReference/ApiReference-query-DescribeInstances.html|c
instances]]**: Outputs a list of launched images belonging to one particular user,
[[http://docs.amazonwebservices.com/AWSEC2/2009-04-04/APIReference/ApiReference-query-RunInstances.html|check
this link for the method description]].
```

- **terminate instances**: Shutdown a virtual machine(or cancel, depending on its state), check this link for the metho

3.2.1 User Account Configuration

An account is needed in order to use the OpenNebula cloud. The cloud administrator will be responsible for assigning these accounts, which have a one to one correspondence with OpenNebula accounts, so all the cloud administrator has to do is check the configuration guide to setup accounts, and automatically the OpenNebula cloud account will be created.

In order to use such an account, the end user can make use of clients programmed to access the services described in the previous section. For this, she has to set up his environment, particularly the following aspects:

- **Authentication**: This can be achieved in three different ways, here listed in order of priority (i.e. values specified in the argument line supersede environmental variables)
  - Using the **commands arguments**. All the commands accept an **Access Key** (as the OpenNebula username) and a **Secret Key** (as the OpenNebula password)
  - Using **EC2_ACCESS_KEY** and **EC2_SECRET_KEY** environment variables the same way as the arguments
  - If none of the above is available, the **ONE_AUTH** variable will be checked for authentication (with the same used for OpenNebula CLI).
- **Server location**: The command need to know where the OpenNebula cloud service is running. That information needs to be stored within the **EC2_URL**[2] environment variable (in the form of a http URL[2], including the port if it is not the standard 80).

The `EC2_URL`[2] has to use the FQDN of the EC2-Query Server

3.2.2 Hello Cloud!

Lets take a walk through a typical usage scenario. In this brief scenario it will be shown how to upload an image to the OpenNebula image repository, how to register it in the OpenNebula cloud and perform operations upon it.

- **upload_image**

Assuming we have a working Gentoo installation residing in an **.img** file, we can upload it into the OpenNebula cloud using the **econe-upload** command:

The user should take note of this **ImageId**, as it will be needed to register the image.

- **register_image**

The next step should be registering the image to enable its instantiation. We can do this with the **econe-register** command:

- **describe_images**

We will need the **ImageId** to launch the image, so in case we forgotten we can list registered images using the **econe-describe-images** command:

* **run_instance**

Once we recall the ImageId, we will need to use the **econe-run-instances** command to launch an Virtual Machine instance of our image:

We will need the **InstanceId** to monitor and shutdown our instance, so we better write down that 15.

- **describe_instances**

If we have too many instances launched and we don't remember everyone of them, we can ask **econe-describe-instances** to show us which instances we have submitted.

We can see that the instances with Id 15 has been launched, but it is still pending, i.e., it still needs to be deployed into a physical host. If we try the same command again after a short while, we should be seeing it running as in the following excerpt:

* **terminate_instances**

After we put the Virtual Machine to a good use, it is time to shut it down to make space for other Virtual Machines (and, presumably, to stop being billed for it). For that we can use the **econe-terminate-instances** passing to it as an argument the **InstanceId** that identifies our Virtual Machine:

You can obtain more information on how to use the above commands accessing their Usage help passing them the **-h** flag