HOT Template Validator

Telefonica tool for template verification

Table of Contents

[Scope 2](#_Toc10803216)

[Instance configuration 2](#_Toc10803217)

[HTV installation and configuration 3](#_Toc10803218)

[Installation 3](#_Toc10803219)

[Directory structure 4](#_Toc10803220)

[Yaml files upload 5](#_Toc10803221)

[Crontab configuration 5](#_Toc10803222)

[Syntax analysis 5](#_Toc10803223)

[Nomenclature files 6](#_Toc10803224)

[Log file format 6](#_Toc10803225)

# Scope

This document is intended to describe in detail the creation of a Linux instance or docker application, which allows the execution of the software that will perform the syntactic analysis of the yaml files sent by the vendors and the compatibility with the UNICA environment. This will allow to have a higher percentage of successes during the deployment of VNF applications and to have a faster interaction with the vendors.

# Instance configuration

A Linux installation will be used to run the HOT Template Validator application, from now on HTV.

It is assumed the installation of a Linux Ubuntu 16.04 or greater. For the installation we will use an Ubuntu server image. The only package to be added during the installation will be the ssh service.

Initially, only one user will be created, "devuser", which will configure the environment and install the HTV application.

He must have permissions to use sudo.  
The instance must reside on a subnet that allows access to the internet.

After the installation, log in to the instance with root user.

Update the list of available packages

*# apt-get -y update*

Install GNU compiler library (GCC), Python, and the dependent libraries (currently, the clients do not support Python 3.)

*# apt-get -y install gcc libffi-dev libssl-dev python python-dev python-pip*

*python-virtualenv virtualenvwrapper*

On the Ubuntu server a devstack package can be used to install OpenStack all-in-one environment.

Following the procedure:

*# apt-get install git-core*

*$ git clone https://opendev.org/openstack/devstack.git*

The DevStack master branch generally points to trunk versions of OpenStack components. For older, stable versions, look for branches named stable/[release] in the DevStack repo. For example, you can do the following to create a juno OpenStack cloud:

*$ git checkout stable/juno*

Create the file local.conf in devstack directory with the following content:

[[local|localrc]]

HOST\_IP=X.X.X.X # set this to your IP

FLAT\_INTERFACE=xxx # change this to your eth0

DATABASE\_PASSWORD=password

RABBIT\_PASSWORD=password

SERVICE\_PASSWORD=password

SERVICE\_TOKEN=password

ADMIN\_PASSWORD=password

enable\_plugin neutron https://git.openstack.org/openstack/neutron

enable\_plugin heat https://opendev.org/openstack/heat stable/master

enable\_plugin heat-dashboard https://git.openstack.org/openstack/heat-dashboard

ENABLED\_SERVICES=rabbit,mysql,key

ENABLED\_SERVICES+=,g-api,g-reg

ENABLED\_SERVICES+=,n-cpu,n-api,n-crt,n-obj,n-cond,n-sch,n-novnc,n-cauth

ENABLED\_SERVICES+=,neutron,q-svc,q-agt,q-dhcp,q-meta,q-l3

ENABLED\_SERVICES+=,cinder,c-sch,c-api,c-vol

ENABLED\_SERVICES+=,horizon,h-eng,h-api,h-api-cfn,h-api-cw,heat-dashboard

enable\_plugin ceilometer https://git.openstack.org/openstack/ceilometer

enable\_plugin aodh https://git.openstack.org/openstack/aodh

CINDER\_LVM\_TYPE=thin

CINDER\_VOLUME\_CLEAR=none

VOLUME\_BACKING\_FILE\_SIZE=30G

# -----

DEFAULT\_VOLUME\_GROUP\_NAME=stack-volumes-default

enable\_service s-proxy s-object s-container s-account

SWIFT\_HASH=66a3d6b56c1f479c8b4e70ab5c2000f5

SWIFT\_REPLICAS=1

SWIFT\_DATA\_DIR=$DEST/data

enable\_service ceilometer-acompute ceilometer-acentral ceilometer-anotification ceilometer-collector ceilometer-alarm-evaluator, ceilometer-alarm-notifier ceilometer-api

# Enable LBaaS v2

enable\_plugin neutron-lbaas https://git.openstack.org/openstack/neutron-lbaas stable/mitaka

enable\_plugin octavia https://git.openstack.org/openstack/octavia

enable\_service q-lbaasv2

enable\_service octavia

enable\_service o-cw

enable\_service o-hm

enable\_service o-hk

enable\_service o-api

# Octavia

OCTAVIA\_AUTH\_VERSION=3

#DOWNLOAD\_DEFAULT\_IMAGES=False

# disabled services

disable\_service n-net

# HTV installation and configuration

## Installation

For the installation of HTV follow the steps below:

The application will be installed in the home of the user “devuser”, therefore in order to download the application execute this command from the terminal directly in devuser’s home:

(the user has to be ‘invited’ into the bitbucket repository, create an Atlassian account and save his public ssh key into the Atlassian account).

*$ git clone git@bitbucket.org:iumii/openstack-script.git*

After the download from the repository a directory will appear “yamlvalidator” in devuser’s home.

Change directory from terminal and enter *./yamlvalidator.*

*$ cd /home/devuser/yamlvalidator*

Execute the *setup.sh* file in order to update python3, pip, virtualenv, create and activate the virtual environment and install the requirements for the application:

*$ ./setup.sh*

After the requirements are installed it is necessary to activate the virtual environment in order to have the python scripts running:

*$ source ./venv/bin/activate*

There are few simple last steps to follow in order to correctly use YamlValidator:

* Move the openrc files inside the .*/rc\_files* directory
* Move the yaml files inside the .*/TemplateLocalStorage* directory (for details read the following chapter “Yaml files upload”

After these steps it is possible to run *shadow.py* command; it will ask the operator for a password for each openrc file:

*$ ./shadow.py*

Remember: this command must be executed again necessarily if the Openstack server passwords change or if the openrc files change.

Furthermore it will also generate a list of crontabs in the *list\_cron.txt* file that it is possible to use in the general crontab file. I twill generate a crontab for each *openrc.sh* file.

At this point everything is set and the application can run correctly.

It is possible to run it in through an interactive and no interactive way:

* Interactive:

*$ python validator.py*

It will ask for which openrc file to choose and then it will examine every yaml file inside the *./TemplateLocalStorage* directory, giving a detailed report on each file in the terminal.

* No interactive (two ways):

1. From terminal, where “*admin-openrc-V3.sh”* is our openrc file:

*$ python validator.py –-openrc=”admin-openrc-V3.sh”*

1. Or via crontab job (the crontabs are automatically generated by *shadow.py*, and saved into *./list\_cron.txt* file, it is necessary to copy and paste them into the main system crontab file). In the example the password is the result of the encryption of the openstack password for that openrc file:

***\*/10 \* \* \* \* source /home/devuser/YamlValidator/venv/bin/activate && python /home/devuser/YamlValidator/main.py --password=”b’asufhaisufh234rhwed==’” --openrc="admin-openrc-V3.sh"***

## Directory structure

In the home of the user "devuser" the following directory structure will be created:

*/home/devuser/yamlvalidator/*

*./TemplateLocalStorage*

*./ValidYamlFiles*

*./WarningYamlFiles*

*./ErrorYamlFiles*

*./Log*

Where */home/devuser/yamlvalidator* is the root of the software.

If the application folder will be moved from its original path it is important to execute this command:

*$ python cron\_update.py*

In order to re-generate the crontabs with the right path.

While if the openrc files change or the passwords for them change is important to execute *shadow.py.* so that the application saves the passwords again:

*$ python shadow.py*

## Yaml files upload

The upload of the files must take place with a secure protocol, thus sftp or scp.  
The yaml files must be stored in the *./TemplateLocalStorage* directory in order to be analyzed by the script, therefore the whole path for scp will be for example:

*$ scp /PATH-VENDORS/yaml\_files devuser@server\_ip:/home/devuser/yamlvalidator/TemplateLocalStorage/*

If the files will be saved somewhere else the script will not be able to take them for the validation.

## Crontab configuration

The Linux crontab must be configured in order to execute the application regularly. The parameters to be considered are the application path and the scheduling. The application must be performed with the command plus the arguments –password and --openrc:

*/PATH-BIN/python /PATH-APP/validator.py*

And it must be performed every 10 minutes (TBD).

Example of a crontab line:

***\*/10 \* \* \* \* source /home/devuser/YamlValidator/venv/bin/activate && python /home/devuser/YamlValidator/main.py --password=”b’asufhaisufh234rhwed==’” --openrc="admin-openrc-V3.sh"***

The application will automatically create the crontabs after the execution of the command *shadow.py* and it will be possible to find those crontab lines inside the file *./list\_cron.txt.*

## Syntax analysis

The first step of file analysis of the script concerns the syntax.

We will distinguish 3 events during the running of the script:

1. The file is syntactically correct
2. The file contains warnings
3. The file contains errors

Event 1 → If the analyzed file contains no errors it is moved to the subdir *./ValidYamlFiles .*

Event 2 → If the file contains warnings, it is possible that it is due to syntax errors or a non-standard yaml format, but that does not affect the operation of the template, that means the files are still valid for the Heat client, therefore the files with warnings will be moved to the subdir *./WarningYamlFiles*. For every file with warnings a log will be created containing the warning messages and it will be moved into the dir *./Log*. These yaml files with warnings can be sent to UNICA or returned to the vendor within the log. This will depend on the operator’s choice.

Event 3 → If the file contains errors it means that it is not a valid yaml file or valid heat template for many reasons, it will then be moved to the subdir *./ErrorYamlFiles* and it must be returned to the vendor together with the log file. For every file with errors a log will be created containing the errors messages and it will be moved into the dir *./Log*.

## Nomenclature files

The log files generated by the tool will have the following format as file name:

Error Log → *<yaml-file-name>-<TimeStamp>-error.log*

Warnign Log → *<yaml-file-name>-<TimeStamp>-warning.log*

## Log file format

Below is a description of the contents of the files generated by the tool in the event of errors and warnings. The error file will have a format similar to the following example, you will find lines with the reference to the error and the exact position of the error (line - column):

*ScannerError: while scanning for the next token*

*found character '%' that cannot start any token*

*in "<string>", line 15, column 231:*

*... CKET\_BRANCH, \"Date\": $(date +"%m-%d-%y"), \"Time\": $(date +"% ...*

The warning file will have a format similar to the following example, you will find lines with references to warning messages and the exact position of the warning (line - column):

*5:1: missing document start "---" (document-start)*

*12:1: wrong indentation: expected 2 but found 0 (indentation)*

*94:15: too many spaces inside brackets (brackets)*

*94:24: too few spaces after comma (commas)*

*94:31: too many spaces inside brackets (brackets)*

*104:81: line too long (92 > 80 characters) (line-length)*

*117:14: too many spaces inside braces (braces)*

*117:42: too many spaces inside braces (braces*

**Immagine che contiene mappa

Descrizione generata automaticamente**

Figure 1 YamlValidator schema