GENI - Virtual Topology Service Documentation

Release 1.0

Nick Bastin

Mar 02, 2018

Contents

Welcome to the online documentation for the VTS deployment in the NSF GENI federation.

The Virtual Topology Service (VTS) is a high-level orchestration system for creating complex network topologies for networking research and classroom exercises. Despite the name, not all VTS topologies are virtualized - as covered in detail in the documentation the system will orchestrate your requested topology on the infrastructure that is available, some of which may be virtualized while parts may be implemented by giving you direct control of existing infrastructure.

Contents:

CHAPTER 1

High Level Concepts

1.1 Circuit Planes

1.2 Function Surfaces

CHAPTER 2

GENI Details

2.1 Images

2.1.1 bss.ovs

2.1.2 bss.quagga

2.1.3 uh.simple-dhcpd

2.1.4 uh.cn4421

CHAPTER 3

How-To Examples

3.1 Associate Your Dropbox Account with a VTS Site

This quick tutorial will show you how to associate a Dropbox account with a VTS site, which will enable you to upload data to your Dropbox from within your isolated slivers.

You must associate your account with *each* site you want to upload to Dropbox from, but you do not need to subsequently associate slivers created at those sites. The Dropbox account is bound to your GENI user credential, and other users using your slices cannot upload to your account (although they can upload data to their own accounts).

This association process requires the use of `geni-lib`, as no other tools have support for the non-standard API that VTS uses for this process.

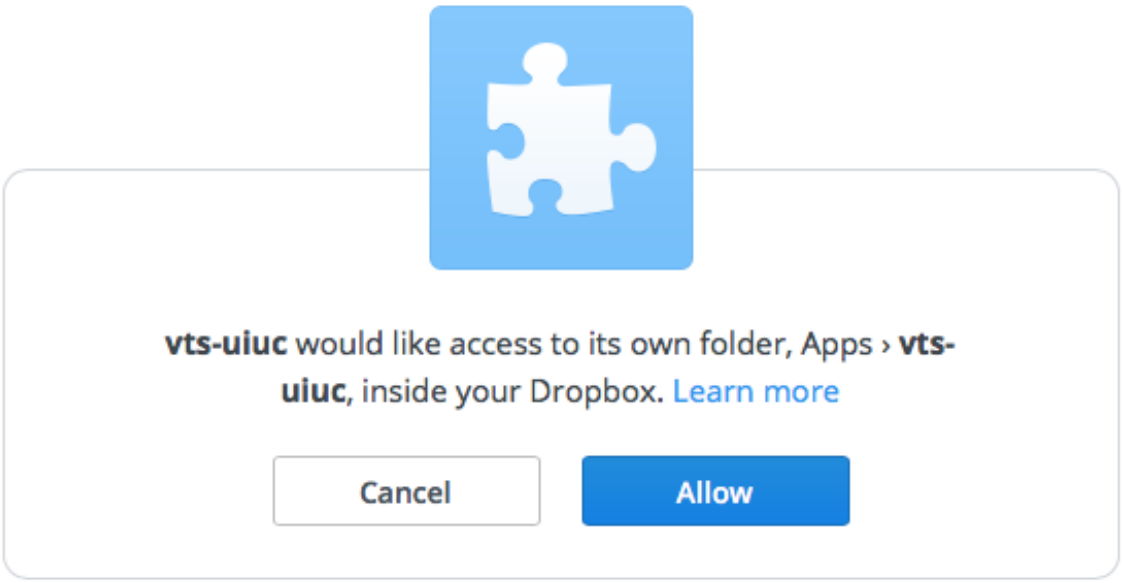
3.1.1 OAuth2 Dropbox Authorization

This is a two-stage process, where you will get a Dropbox authorization URL from a given VTS site, and then after visiting that web page in your browser (and logging into Dropbox as necessary), you will be given a code from Dropbox that you will send back to the VTS site:

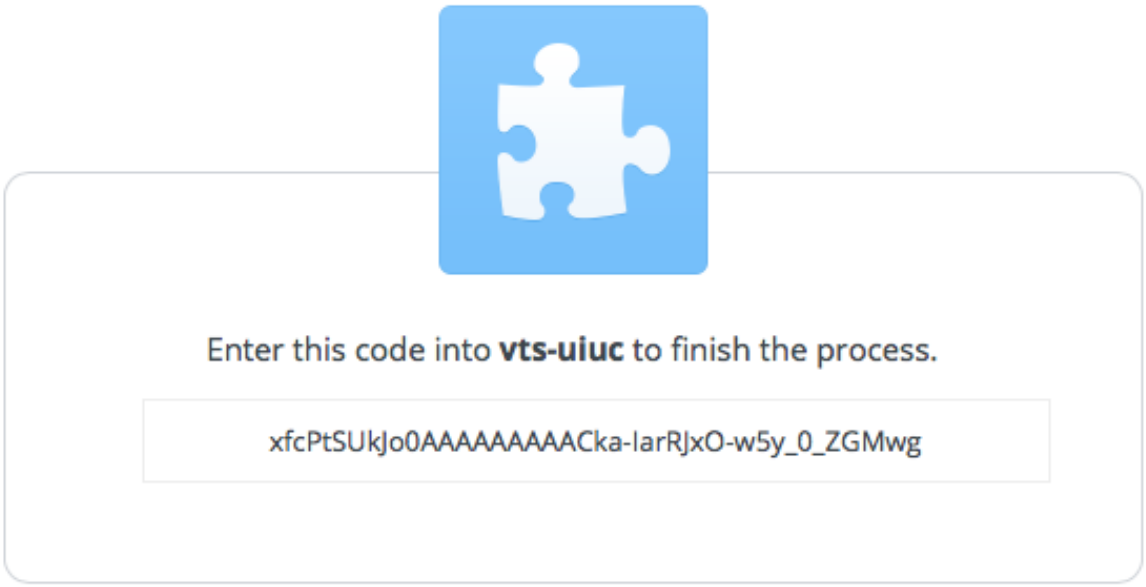
```
>> SITE = VTSAM.Illinois

>> SITE.dropboxLink(context)
{'authorize_url': 'https://www.dropbox.com/oauth2/authorize?response_type=code&client_
↪id=poczslfi15gp4l9'}
```

Take the URL value from `authorize_url` and paste it into your web browser, where you should get a page that is similar to the image below (the names will differ based on the VTS site you have chosen):



After you click the *Allow* button, you will be presented with a code to send back to VTS to finalize the authorization process:



Now, copy that code back to `geni-lib` and finalize your Dropbox authorization for the site you have chosen:

```
>> SITE.dropboxFinalize(context, "xfcPtSUkJo0AAAAAAAAACka-IarRJxO-w5y_0_ZGMwg")
{}
```

Note: A successful response is an empty `{}` - any failure should result in an exception.

This concludes the association and now any resource reservations you make that include Dropbox mounts can send the contents of those directories to your Dropbox account. You can also upload from any existing slivers you have access to that have already reserved Dropbox mounts at this site.

Uploaded files will appear in your Dropbox account under the "Apps" directory, indexed by the site, sliver URN, host client_id, and host mount path. VTS cannot write to other locations in your Dropbox, and you cannot send files to your slivers from these directories (the functionality is upload-only from VTS).

3.2 Create A New Container Image

Most vertices in VTS topologies are implemented as containers on unix-like systems (including hardware forwarding devices such as Pica8 and Znyx switches). This How-To describes the basic components of these images, and how to get started in creating new ones.

This How-To does not cover creating the synthetic images used on vendor hardware that does not expose a unix-like environment.

3.2.1 High Level Pieces

Every VTS container image is composed of a maximum of 3 distinct components:

- A disk image used to launch each instance of the container
- A JSON-based spec file describing the image usage to the VTS orchestrator
- (optional) A python-based handler allowing for fine-grained integration into the orchestrator

3.2.2 Disk Images

VTS Disk Images are typically built using Dockerfiles (although they are not *run* using Docker). This allows images to be created very quickly using existing base OS images available from Docker, and merely adding the software required for the function you are adding. While there is no officially 'blessed' directory layout used for image creation, most image builds use the following layout:

```
/my-image/
          build/
                Dockerfile
                ...
          runtime/
                  spec.json
```

There is a skeleton image build directory (in *skel/*) available in the UH-NetLab Images repository, which you can copy to get a basic image build environment. The skeleton image uses Alpine linux as the base, and includes supervisor, ssh, and rsyslog which enables support for standard orchestrator services. The *install.py* script in the same repository will install images built in the same format as the skeleton, using a process well understood by VTS administrators.

Once you have a skeleton layout, edit your *Dockerfile* and *supervisord.conf* as necessary (and add any additional required files) to describe the disk image your container requires.

Note: Since the Docker tools are only used to build images, but not to run them, you can only use features of Dockerfile that are used during the build process. Features such as EXPOSE and VOLUME are runtime values that will not be evaluated. As a result, it is generally better to refer to pre-existing image repositories for examples rather than going to the Docker documentation.

3.2.3 Spec Files

JSON specification files are used to provide important metadata about your image to the orchestrator (how much memory to allocate, attributes that may be supplied by the user at reservation time, etc.). The spec files are documented at *Container Image Specfiles*. You can also review other examples in existing repositories.

3.2.4 Handlers

Image Handlers are Python classes subclassed from *foam.vts.images.ImageHandler* that are added at runtime to the orchestrator in order to provide programmatic handling for instances that use your image. This generally is limited to two entrypoints:

- Callbacks when instances using your image are being built
- PerformOperationalAction (POA) API extensions

Note: The UH-Netlab image repository contains a number of useful handler subclasses in *vts_uh.bases* that you may want to subclass from instead of using *foam.vts.images.ImageHandler* directly.

Each handler can specify a number of callbacks that are invoked at various points during the image instantiation process. There is currently limited documentation for these callbacks, although there is a wealth of example handler code. The current callbacks are:

- *.prebuild (self, cobj)* - Invoked before a container using your image has been built (for each requested instance)
- *.postlocaltopo (self, cobj)* - Invoked after a container using your image has been instantiated, and has local networking (all interfaces have been attached).
- *.postnettopo (self, cobj)* - Invoked after the entire topology graph creation is complete (although you cannot guarantee that *postnettopo* has been called for any other instances, as no order is specified)
- *.geniSetup (registry)* (static method) - Invoked *once* (per process) for each image spec that references this handler. Used to set up new POA endpoints and any other one-time initialization items.

While all of these methods take the *cobj* argument, it should not be used and *self.container* should be used instead.

3.2.5 POA Extensions

Generally POA extensions should utilize the *vts_uh.bases._wrapPOA* wrapper function which provides a consistent interface to tools for new POA endpoints. The *vts_uh.bases.AlpineHost* handler class has clean examples of how to use this wrapper.

3.3 Install VTS into an Ubuntu Server 16.04 Xenial Xerus

This quick tutorial will show you how to create your own private VTS site, by installing the VTS software in your own ubuntu box. Note that this procedure below has been verified to work for a fresh installation of the Ubuntu Server 16.04 LTS Xenial Xerus. Other versions, and operating systems may require some additional steps.

3.3.1 Installation Procedure

1. On a freshly installed Ubuntu Server 16.04 LTS Xenial Xerus, run the command:

```
# sudo apt-get update
```

2. download the vts installation script:

```
# wget https://bitbucket.org/nbastin/vts-install/freshinstall.py
```

3. run the command to install:

```
# ./freshinstall.py
```

4. respond to the installation prompts as suits your purpose. However, select yes when prompted, for the images you'll need for your experiments

After installation you can use your newly installed VTS from your genilib environment, by set your aggregate using the command:

```
# am = VTSAM.VTS("servername", "ip_address")
```

Then you can use the 'am' to make calls to the VTS instance. For example:

```
# am.listresources(context)
```

Recommended next step: configure your VTS installation to support drop box. *Configure dropbox for your private VTS installation*

3.4 Configure dropbox for your private VTS installation

This quick tutorial will show you how to enable dropbox on your own private VTS installation. This will allow hosts on networks you create to be able to upload files to linked dropbox accounts.

3.4.1 Configuration Procedure

1. Ensure that you have successfully installed vts according to the directions at: *Install VTS into an Ubuntu Server 16.04 Xenial Xerus*
2. Create a dropbox app at: https://www.dropbox.com/developers then generate an "access token" and a "secret" which you will need in order to complete the dropbox configuration within your VTS box
3. Within your VTS box, locate the password for your vts in /etc/foam.passwd. you can use the command:

```
# sudo cat /etc/foam.passwd
```

4. Run the commands below in your vts server, remember to replace "your_access_token" and "your_access_secret" with actual values you generated in step 2 above. In addition, whenever you are prompted for a password, use the password you obtained from step 3.:

```
# cd /opt/foam/mount
# sudo foamctl host:build-mount --name=dropbox-user --size=2000
# cd ~
# foamctl config:set-value --key=dropbox.user-vols.root --value=/opt/foam/mounts/
↪dropbox-user
# foamctl config:set-value --key=dropbox.app-key --value=my_access_token
# foamctl config:set-value --key=dropbox.app-secret --value=my_access_secret
```

5. Restart the foam service:

```
# systemctl restart foam
```

Note that you will need to associate your dropbox account with the VTS installation, using your genilib environment, by following the directions at *Associate Your Dropbox Account with a VTS Site* you may need to replace the first command in that guide with:

```
am = VTSAM.VTS("servername","ip_address")
am.dropboxLink(context)
```

3.5 Install custom images in your private VTS installation

This quick tutorial will show you how to install VTS images within your own private VTS installation. You can follow the steps to install existing VTS images, or new images that you have created yourself, by following the directions at *Create A New Container Image*

3.5.1 Installation Procedure

1. Ensure that you have successfully installed vts according to the directions at: *Install VTS into an Ubuntu Server 16.04 Xenial Xerus*
2. From your VTS box, download the VTS images intaller script:

```
# hg clone https://oadele3@bitbucket.org/UH-netlab/vts-images/install.py
```

3. To install your own custom image, first ensure that your custom image has the directoy structure as described in *Create A New Container Image*. That is it must have the 'build' and the 'runtime' subdirectories within it.
4. Copy your custom image's directory into the same directory to which you downloaded the image installer script You can use the command below to copy.

```
# cp /path/to/your/image .
```

5. To install the new image, run the command:

```
# ./install.py image_dir_name
```

   where image_dir_name is the directory name of the image you want to install

6. Restart the foam service:

```
# systemctl restart foam
```

CHAPTER 4

User Tutorials

4.1 Simple L2 Switching

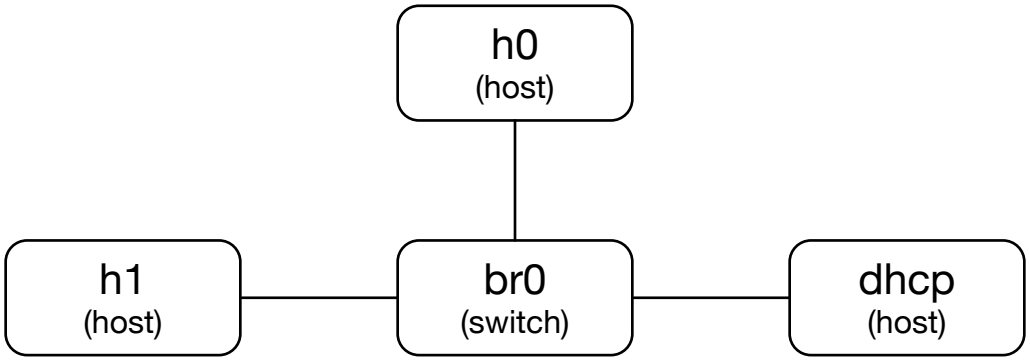
This is a brief walk-through of a basic VTS request that will create a topology with several hosts and a DHCP server. This topology will serve as the foundation for several other tutorials that you may also execute, walking you through basic VTS workflows as well as a simple illustration of the power of full control of isolated network topologies.

This tutorial may be executed using the `geni-lib` suite (see here for installation instructions), either the `genish` command line tool, or the identically named *ipython* extension.

For both tools you will need to download your `omni.bundle` file from the GENI Portal. You can find instructions for doing so in the geni-lib docs.

You can also use VTS with a Cloudlab account (without an *omni.bundle*), and you can find documentation for acquiring these credentials in the geni-lib docs. If using Cloudlab credentials with `geni-lib`, please review the geni-lib documentation for using these credentials.

4.1.1 Topology



4.1.2 Build the Request Object

In order to request this topology, we can create a resource request object for VTS using `genish` (at the command line or in *ipython* notebooks):

```
r = VTS.Request()
```

We then add our switch:

```
br = r.Datapath(VTS.OVSL2Image(), "br0")
```

There are a limited number of switch images available in GENI - the two most commonly used are the one above, which is a normal MAC-learning bridge with VLAN support, and an Openflow-supporting image (referenced in `geni-lib` as `VTS.OVSOpenFlowImage`, with some required configuration parameters). The second argument to `Datapath` is the client_id of the bridge, which you will need to know later in order to query it for monitoring and state information. We assign the return from `r.Datapath(...)` to a variable that we will use later in the request to build links.

Now we add our hosts:

```
h0 = r.Container(VTS.Image("uh.net-client"), "h0")
h1 = r.Container(VTS.Image("uh.net-client"), "h1")
dhcp = r.Container(VTS.Image("uh.simple-dhcpd"), "dhcp")
```

Switch images are added using the `Datapath` constructor, while hosts are added using the `Container` constructor. As with datapaths there are two arguments - the image object to be used, and the client_id of the resultant host. In the case of hosts, the client_id is also used as the hostname, so it's preferable to not choose characters that would not be valid in a hostname.

In this case we use two images that are available at all GENI VTS sites - the "uh.net-client" image, which is a basic linux host with networking tools installed (tcpdump, scapy, iperf, etc.), and the "uh.simple-dhcpd" image, which is a trivial DHCP server that supports a single IPv4 subnet. By default the subnet used for DHCP is `192.168.50.0/24`, but you can change that value by setting the image attribute `subnet` if you desire:

```
dhcp.image.setImageAttribute("subnet", "10.70.10.0/23")
```

Now we need to add links between our nodes:

```
VTS.connectInternalCircuit(br, h0)
VTS.connectInternalCircuit(br, h1)
VTS.connectInternalCircuit(br, dhcp)
```

Now our request is complete, and we can send it to the aggregate manager at a site to reserve our resources.

4.1.3 Make the Reservation

Choose a site, and store it in a short-to-type variable name for later:

```
SITE = VTSAM.StarLight
```

You may also want to store your slice name in a short variable name for ease of use or changing later:

```
SLICE = "my-slice-name"
```

Of course, change that value to the name of a slice that you control.

Now create the sliver for our resource request, and get the manifest back:

```
manifest = SITE.createsliver(context, SLICE, r)
```

If there is an exception, note the problem and either resolve it yourself (if you need to delete an existing sliver, etc.), or send email to `geni-users@googlegroups.com` to see if someone can help you. Otherwise, proceed with the rest of the tutorial using the manifest that was returned above.

4.1.4 Visualize Reserved Resources

If you are using the `genish` *ipython* notebook extension, you can get a visual topology diagram using the built-in manifest renderer:

```
genish.showtopo(manifest)
```

If you are using the command line `genish` shell, you can get dot output, which you can use in *graphviz* or any other *dot* supporting tool:

```
util.builddot([manifest])
```

Note: `util.builddot` takes a list of manifests, so if you only have one manifest you still need to put it into a list.

4.1.5 Connect to Topology Hosts

Now that the hosts in this topology have been provisioned, you can get login information in order to configure SSH to access your hosts:

```
genish.printlogininfo(manifest)
```

If you are using the `genish` shell, you can access a similar method in the `util` module:

```
util.printlogininfo(manifest=manifest)
```

Note: The core `util.printlogininfo` function takes a number of arguments that can be used to get a manifest that you do not already have. Given that you already have a manifest, you can pass it in directly using the `manifest` argument.

The output comes in 4 pieces - the client_id you requested in your reservation, the username you should use to access this resource, the internet-accessible hostname or IP you can use to reach the resource, and the port number to be used for SSH. It will come in a table in *ipython*, or in a simple list in the command line output:

```
[h0][ywauusshw5k] starlight.vts.bsswks.net: 22
[h1][k10172ltax6] starlight.vts.bsswks.net: 22
```

In this case, if we want to connect to the container representing `h0`, we have the information that we can give to an SSH client to connect:

```
ssh -i ~/.ssh/id_geni_ssh_rsa ywauusshw5k@starlight.vts.bsswks.net
```

Note: Since the default SSH port is 22, we don't have to provide the port to the client. Also as with all GENI slivers, the SSH public key referenced in your `context` object is the one used for authentication. By default it is the one used above.

Now that you have a connection to the host, you can run `dhclient` to get an IP address from the DHCP server that you provisioned in your topology:

```
/ # dhclient -v eth1
```

As we asked for verbose output, you should see something like the output below:

```
Internet Systems Consortium DHCP Client 4.3.4
Copyright 2004-2016 Internet Systems Consortium.
All rights reserved.
For info, please visit https://www.isc.org/software/dhcp/

Listening on LPF/eth1/f6:5e:13:3c:0f:7c
Sending on   LPF/eth1/f6:5e:13:3c:0f:7c
Sending on   Socket/fallback
DHCPDISCOVER on eth1 to 255.255.255.255 port 67 interval 3
DHCPREQUEST on eth1 to 255.255.255.255 port 67
DHCPOFFER from 10.70.10.1
DHCPACK from 10.70.10.1
bound to 10.70.10.10 -- renewal in 375 seconds.
```

Now that you have established that your topology is working, you can walk through the below sections to view internal network state.

4.1.6 View DHCP Lease State

Many images offer a number of POA (*Perform Operational Action*) APIs that can be used to gather data about the current image state. In the case of the `uh.simple-dhcpd` image we can gather information about the leases that the DHCP server has currently allocated:

```
SITE.getLeaseInfo(context, SLICE, "dhcp")
```

This will give you a table with the current lease state - allocated IP addresses, MAC addresses, when they expire, and (depending on information offered by the host) the hostname of the requesting host. As long as `dhclient` continues to run on each host the client process will renew the leases as necessary.

4.1.7 View Switch MAC Table

Similar to the information offered by the `uh.simple-dhcpd` image, we can inspect the switch state to view learned MAC addresses:

```
SITE.dumpMACs(context, SLICE, "br")
```

This will give you table with the current MAC table state on the switch (you can also pass it a list of switches, if your topology has more than one switch). This will include the port number the MAC is learned on, the VLAN if appropriate, the MAC itself, and the current age. The default aging time for the OVSL2Image is 300 seconds, so a MAC not seen for 5 minutes will expire from the table.

CHAPTER 5

Site Administration

5.1 Container Images

The runtime details of Linux container images (regardless of engine) are dictated by spec files in the *_basespecs* repository.

A typical GENI VTS installation will have a repository that is similar to below:

```
/opt/foam/docker-imagespecs/_basespecs/
├── bss.quagga
│   └── spec.json
├── uh.cn4421
│   └── spec.json
├── uh.dnsdhcp
│   └── spec.json
├── uh.dnsroot
│   └── spec.json
├── uh.pynfdev
│   └── spec.json
└── uh.simple-dhcpd
    └── spec.json
```

A good source of publicly available images is the University of Houston image repository on bitbucket.

The specfile format is defined in the *developer documentation*, and basic parameters such as memory bounds and custom handlers can be easily modified for a given site installation.

CHAPTER 6

Developer Documentation

6.1 Container Image Specfiles

Container runtime parameters are defined by JSON spec files. Parameters are introduced over time, and the version of the file changes as a result. All versions of the file are supported at runtime - the version specified outlines the parameters that will be found in that file.

Note: If you wish to specify a parameter that was introduced in a given version of the file, you must also specify the required parameters for all previous versions.

6.1.1 Version 1

Parameter	name
Required	Yes
Type	String
Description	Image name advertised to clients
Parameter	image-name
Required	No
Type	String
Description	Image name used by instance runtime. Defaults to *name* if unspecified.
Parameter	max-slots
Required	Yes
Type	Integer
Description	This parameter is ignored, but required.
Parameter	memory
Required	Yes
Type	Integer
Description	The default amount of memory in megabytes allocated to containers using this image

6.1.2 Version 2

Parameter	console-shell
Required	Yes
Type	String
Description	The path (internal to the instance) of the shell to run when a user requests a console. May be empty to disable console access.

6.1.3 Version 3

Parameter	capabilities
Required	Yes
Type	List
Description	A list of linux capabilities to set for instances using this image. Pre-version-3 instances are granted the set of capabilities specified by the list ['NET_ADMIN', 'NET_RAW'].
Parameter	volumes
Required	Yes
Type	List
Description	Non-ephemeral volumes added to instances using this image. May be empty.
Parameter	network-shadow
Required	Yes
Type	Boolean
Description	Set to *true* if the image **MUST** have the network set up before the entrypoint is invoked. This adds significant overhead to the system, and should be avoided if possible.

6.1.4 Version 4

Parameter	make-user-image
Required	Yes
Type	Boolean
Description	Build a new image based on this one for each requesting user. This is highly discouraged.

6.1.5 Version 5

Parameter	attributes
Required	No
Type	List of Objects
Description	(Review examples in existing images)
Parameter	handler
Required	No
Type	Object
Description	Callback handler class for this image, in the format of: *{"version" : 1, "module" : "<full.module.path>", "class" : "<classname>"}*
Parameter	data-dir
Required	Yes
Type	String
Description	Root directory where volumes are stored for instances using this image. May be empty.

6.1.6 Version 6

Parameter	port-attributes
Required	No
Type	List of Objects
Description	(Review examples in existing images)

6.1.7 Version 7

Parameter	memory-min
Required	No
Type	Integer
Description	The minimum memory a user may request for instances using this image.
Parameter	memory-max
Required	No
Type	Integer
Description	The maximum memory a user may request for instances using this image.

CHAPTER 7

FAQ