Hammr Guide

Release 3.6

UShareSoft

Jun 09, 2017

Contents

This guide contains a complete reference of all features provided by hammr. If you are completely new to hammr, we recommend that you read the *Introduction* and walk through the *Getting Started* section which guides you through how to create your first template, generate a machine image and publish it to a target cloud environment.

Any questions or comments, please get in touch by using the mailing list.

Contents:

CHAPTER 1

Introduction

Hammr is an open source tool for creating machine images for different environments from a single configuration file, or migrating live systems from one environment to another. Hammr is a lightweight client-side tool based on Python, and can be installed on all major operating systems.

A machine image contains a set of operating system packages and other 3rd party software required to run a particular service. Once a machine image is created, it can be used to provision one or more identical running instances. The format of the machine image varies depending on whether you want to run your service on a physical machine (e.g ISO) a virtual datacenter (e.g OVF for VMware vCenter) or cloud environment (e.g. AMI for Amazon EC2).

Hammr can be used as part of your "DevOps tool chain" and in conjunction with other tools such as Jenkins, Chef, Puppet and SaltStack, allowing you to easily build your machine images and maintain your live running instances. Hammr also has migration capabilities, allowing you to scan a live system, generate a machine image for a different environment as well as export it back to a configuration file for sharing.

CHAPTER 2

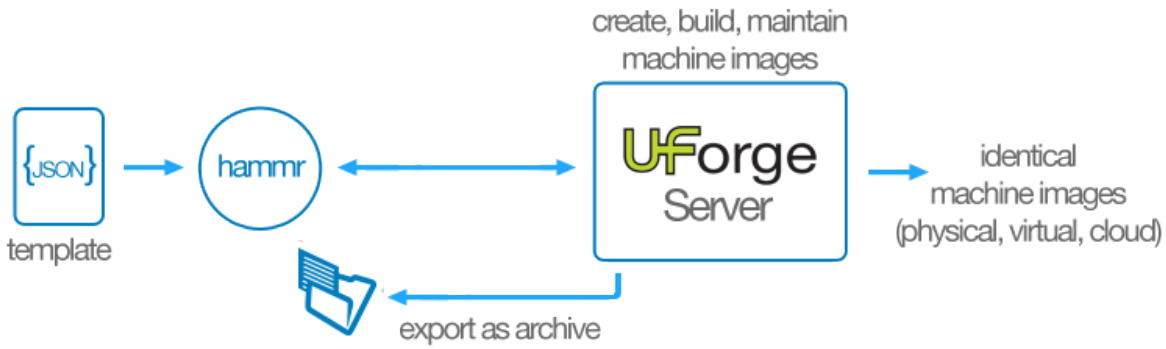
Getting Started

Hammr is a client command-line tool for a UForge Server. A UForge server can be deployed onsite in your own datacenter, or provided in SaaS by a growing number of hosting providers and UShareSoft.

If you don't have your own UForge Server, you can get a free online account here from UShareSoft. Once you have signed up for an account, hammr uses your account to create and manage templates; build machine images; or migrate live systems from one environment to another.

Template Configuration File

All machine images are created from a JSON or YAML configuration file, known as the `template`. Templates provide all the information (os packages, software files, configuration) to describe the machine image you wish to build. This template is used by hammr to create the template (in meta-data) into the UForge Server and build one or more identical machine images. These images can then be registered to the respective environment ready for provisioning. Once a template is created in the UForge Server, hammr can be used to track and apply package updates for the template. Hammr can also export a template registered in the UForge Server to an archive that includes all the software and the original template configuration file.



Installation

To use hammr, you require to install it on the machine you wish to run it. Hammr is based on python, and is supported all major operating systems. The easiest way to install hammr is using `pip` the widely used package management system for installing and managing software packages written in python.

Installing pip

If you already have pip installed on your system, you can skip this step.

To install or upgrade pip, download get-pip.py

Now run the command:

```
$ python get-pip.py
```

For more information on installing pip, please refer to the official pip documentation: http://www.pip-installer.org/en/latest/installing.html

Installing Hammr

Once pip has been installed, you can now install the hammr packages (note, you may have to run this command as `sudo` or `administrator`).

A version of Hammr is compatible with only one version of UForge. To see the compatibility table, go to *Install Compatibility* section.

Please refer to the installation instructions depending upon your desktop type:

For Linux

First of all, you need to install extra packages on your system

Debian based system:

```
$ apt-get install python-dev gcc libxslt1-dev
```

Red-hat based system:

```
$ yum install gcc python-devel libxml2-devel libxslt-devel
```

Now, you are ready to install the latest version of Hammr:

```
$ easy_install progressbar==2.3
$ pip install hammr
```

If you want to install a specific version of Hammr, see *Install Compatibility* to find the compatible version of UForge, and in the code above replace the 'pip install hammr' with:

```
$ pip install hammr==HAMMR-VERSION
```

If you already have hammr installed and want to upgrade to the latest version you can run:

```
$ pip install --upgrade hammr
```

For Mac

For Mac users, you need to have `XCode` installed (or any other C compiler).

You can download the latest version of Xcode from the Apple developer website or get it using the Mac App Store

Run the following command to install the latest version of Hammr:

```
$ xcode-select --install
$ sudo easy_install pip
$ sudo easy_install readline
$ sudo easy_install progressbar==2.3
$ sudo pip install hammr
```

If you want to install a specific version of Hammr, see *Install Compatibility* to find the compatible version of UForge, and in the code above replace the 'sudo pip install hammr' with:

```
$ sudo pip install hammr==HAMMR-VERSION
```

If you already have hammr installed and want to upgrade to the latest version you can run:

```
$ pip install --upgrade hammr
```

For Windows

For Windows users, first install Python 2.7, which can be found here. Download the `msi` file, and install `Python 2.7` by executing the msi file. In the instructions below, we assume that the installation path for Python 2.7 is `C:\Python27`.

Once Python 2.7 is installed, run the command for the latest version of Hammr:

```
c:\Python27> .\Scripts\easy_install.exe hammr
```

If you want to install a specific version of Hammr, see *Install Compatibility* to find the compatible version of UForge, and in the code above replace the '.Scriptseasy_install.exe hammr' with:

```
c:\Python27> .\Scripts\easy_install.exe hammr==HAMMR-VERSION
```

If your Windows does not have a compilation environment, `pycrypto` installation may fail. You can install a `pycrypto` windows binary with this command (change your python version if needed):

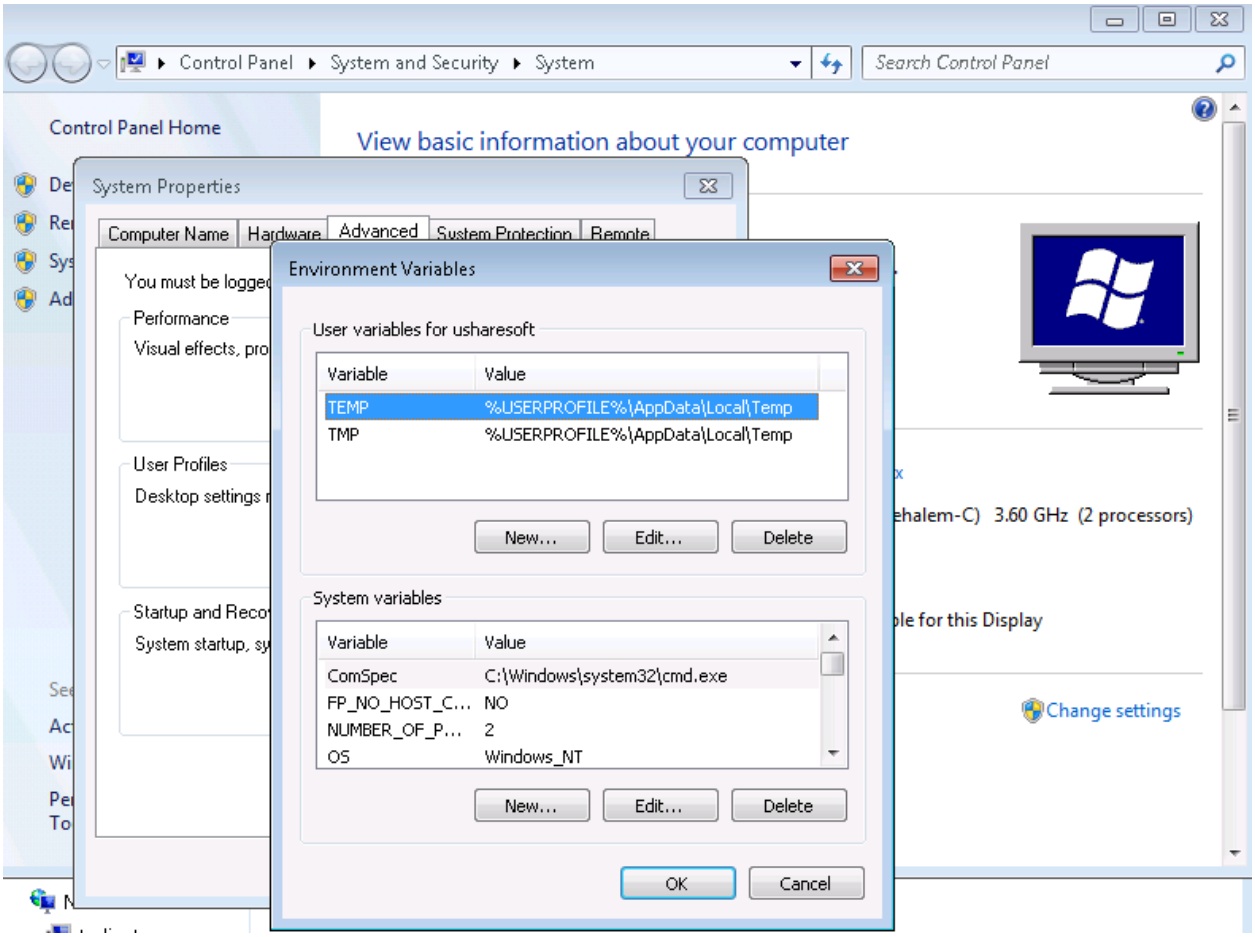
```
c:\Python27> .\Scripts\easy_install.exe http://www.voidspace.org.uk/downloads/
↪pycrypto26/pycrypto-2.6.win32-py2.7.exe
```

If you already have hammr installed and want to upgrade to the latest version you can run:

```
c:\Python27\Scripts> easy_install.exe --upgrade hammr
```

Add Python and hammr to system path: Go to "My Computer > (right click) Properties > Advanced System Settings > Environment Variables"

You will get this configuration window:



In `System Variables`, search for the `Path` variable and click `Edit`. Add the following at the end (replace `C:\Python27` with your Python installation path if it differs):

```
;C:\Python27;C:\Python27\Scripts;
```

From Source

Hammr has a dependency to `uforge_python_sdk`. First, you need to install it:

```
$ pip install uforge_python_sdk
```

or download sources from pypi: https://pypi.python.org/pypi/uforge_python_sdk

Go to the source directory where the `setup.py` file is located. To compile and install, run (as sudo):

```
$ python setup.py build; sudo python setup.py install
```

Now clone the Hammr git repository to get all the source files. Next go to the source directory where the `setup.py` file is located. To compile and install, run (as sudo):

```
$ python setup.py build; sudo python setup.py install
```

This will automatically create the hammr executable and install it properly on your system.

Verifying the Installation

After completing the installation process, to verify that the installation was successful and hammr is available to use, open up a new terminal window and run the command:

```
$ hammr -v
hammr version '0.2.0'
```

Install Compatibility

The following table lists the compatibility between versions of Hammr, UForge_python_sdk and UForge:

Hammr version	UForge_python_sdk version	UForge version
3.7-2	3.7-2	3.7-2
3.7.2-1	3.7.2-1	3.7.fp2-1
3.7-3	3.7-3	3.7-3
3.7.3	3.7.3	3.7.3
3.7.4	3.7.4	3.7.4

If your hammr version is not compatible with the UForge version that you want to reach, hammr will raise an error message with the UForge version:

```
$ hammr --url https://uforge.usharesoft.com/api -u username -p password
ERROR: Sorry but this version of Hammr (version = 'HAMMR_VERSION') is not compatible␣
↪with the version of UForge (version = 'UFORGE_VERSION').
ERROR: Please refer to 'Install Compatibility' section in the documentation to learn␣
↪how to install a compatible version of Hammr.
```

To install the correct version of Hammr, please run the command below indicating HAMMR-VERSION you want:

```
$ pip install hammr==HAMMR-VERSION
```

Launching hammr

Hammr is a command-line tool, allowing you to specify commands that get executed by hammr. Each command may have one or more sub-commands and optional parameters. Hammr provides inbuilt help. To list all the main options, use the `-h`, `--help` flags or `TAB`.

```
$ hammr -h
usage: hammr [-h] [-a URL] [-u USER] [-p PASSWORD] [-v] [cmds [cmds ...]]
To get more information on a sub-command, use the -h, --help flags or TAB for more␣
↪information

$ hammr template -h
================================================================================
Template help
================================================================================
build                          | Builds a machine image from the template
clone                          | Clones the template. The clone is copying the meta-
↪data of the template
create                         | Create a new template and save to the UForge server
delete                         | Deletes an existing template
export                         | Exports a template by creating an archive (compressed␣
↪tar file)
```

```
help                            | List available commands with "help" or detailed help␣
↪with "help cmd".
import                          | Creates a template from an archive
list                            | Displays all the created templates
validate                        | Validates the syntax of a template configuration file
```

Modes

There are three different modes when launching hammr:

- Classic command-line: used in shell scripts or via a terminal
- Interactive mode: where hammr is launched once, providing you a prompt to execute hammr commands
- Batch mode: allowing hammr to execute a series of commands from a file

When using the classic mode, the command `hammr` is used, followed by a command, sub-command and any options. For example:

```
$ hammr os list --url https://uforge.usharesoft.com/api -u username -p password
```

To enter interactive mode, launch the `hammr` command on its own. This provides a prompt, allowing you to enter commands and sub-commands the same way as you would in classic mode.

To use batch mode, create a file containing the list of commands you wish to launch in sequence and then provide this file to hammr via the batch command. For example if you wanted to list all the operating system available to you in batch mode, firstly create a file with the commands to launch, in this case `os list`:

```
$ vi batchfile
os list
```

Launch hammr providing the batch file:

```
$ hammr batch --file batchfile --url https://uforge.usharesoft.com/api -u username -p␣
↪password
os list
Getting distributions for [root] ...
+-----+--------+---------+--------------+---------------------+-----------------+
| Id  |  Name  | Version | Architecture |    Release Date     |    Profiles     |
+=====+========+=========+==============+=====================+=================+
| 121 | CentOS | 6.4     | x86_64       | 2013-03-01 14:01:26 | Server No X     |
|     |        |         |              |                     | Server          |
|     |        |         |              |                     | Minimal         |
+-----+--------+---------+--------------+---------------------+-----------------+
| 87  | Ubuntu | 12.04   | x86_64       | 2012-02-24 19:04:45 | Minimal Desktop |
|     |        |         |              |                     | Server          |
|     |        |         |              |                     | Minimal         |
+-----+--------+---------+--------------+---------------------+-----------------+
```

Authentication

Communication between hammr and the UForge server is done via HTTPS. To send requests to the UForge server, hammr requires the following information:

- UForge Server URL endpoint
- Your account user name

- Your password

This information can be passed to hammr either from command-line options or from a file.

Command-line Parameters

Authentication information can be passed to hammr via command-line options. These options are:

- `-a` or `--url`: the UForge Server URL endpoint. If the URL uses HTTPS, then the connection will be done securely (recommended), otherwise connection will be done via HTTP
- `-u` or `--user`: the user name to use for authentication
- `-p` or `--password`: the password to use for authentication

For example

```
$ hammr os list --url https://uforge.usharesoft.com/api -u username -p password
```

These parameters need to be passed each time you wish to use the command-line.

Using a Credential File

Rather than passing the authentication information as part of the command-line, you can instead store this information in a credential file (`credentials.json` or `credentials.yml`) that will be used every time hammr is launched. Hammr searches for this file in a sub-directory named `.hammr` located in the home directory of the user launching hammr.

Note: If your AppCenter has a self-signed certificate, in order to use hammr with your AppCenter you must use a credentials file.

To use a credential file, go to the `.hammr` sub-directory and create the file `credentials.yml`.

Note: You can also use JSON. In which case you need to create a file `credentials.json`.

```
$ cd ~/.hammr
$ vi credentials.yml
```

Add the authentication and UForge URL endpoint to this file, using the following format:

```
---
user: root
password: password
url: http://10.1.2.24/ufws-3.3
acceptAutoSigned: false
```

If you are using JSON:

```
{
  "user" : "root",
  "password" : "password",
  "url" : "http://10.1.2.24/ufws-3.3",
  "acceptAutoSigned": false
}
```

As this file contains security information, it is recommended to change the permissions on this file, so only you can read or write to it:

```
$ chmod 600 credentials.yml
```

Now every time hammr is launched, you no longer need to provide the authentication information as part of the command-line. Hammr will automatically use the information contained in this file.

Note: The key `acceptAutoSigned` is to accept or not self-signed SSL certificates. Default value is `false`.

Creating Your First Machine Image

Now that hammr is installed, lets build our first machine image. Hammr can be used to build machine images containing pretty much any software for many different environments – from a trusty ISO image for physical machine deployments; to virtual and cloud environments.

In this example we are going to create a nginx machine image for Amazon EC2 based on Ubuntu 12.04 (64 bit).

Note: To go through this tutorial, you are going to need an AWS account. If you don't have one, create a free account here. If you do not wish to create an AWS account, then you can still follow the tutorial, as creating machine images for other environments follows the same basic principles.

There are three phases when creating your machine image:

- Defining the contents of the machine image in a template configuration file
- Generating the machine image from the template to the required environment, in our case Amazon EC2.
- Publishing and registering the image in AWS, ready to provision one or more machine instances from the machine image

The rest of this section highlights these steps to create your first machine image.

Creating the Template

A configuration file, named the template, defines the contents of the machine image and any credential information required to generate and publish the image to the target environment.

Lets create a template for the nginx machine image. Create a file `nginx-template.yml` with the following content.

Note: JSON can also be used.

```
---
stack:
  name: nginx
  version: '1.0'
  os:
    name: Ubuntu
    version: '12.04'
    arch: x86_64
```

```
    profile: Minimal
    pkgs:
    - name: nginx
  installation:
    diskSize: 12288
```

If you are using JSON:

```
{
        "stack": {
                "name": "nginx",
                "version": "1.0",
                "os": {
                    "name": "Ubuntu",
                    "version": "12.04",
                    "arch": "x86_64",
                    "profile": "Minimal",
                    "pkgs": [{
                        "name": "nginx"
                    }]
                },
                "installation": {
                    "diskSize": 12288
                }
        }
}
```

A couple of things to point out at this stage. The `stack` section defines the content of the machine you want to build. There are many sub-sections (see the *Stack* glossary), the:

- `os`: defines the operating system you want to use (in this case Ubuntu 12.04 64bit); the profile type (minimal); and any specific packages to install (nginx)
- `installation`: defines lower level installation parameters. In this example a disk size of 8GB

Now lets create the template using hammr. First lets validate that the configuration file does not have any syntax errors or missing mandatory values, by using the command `template validate` and passing in our template file `nginx-template.yml` (or .json file is you are using JSON).

```
$ hammr template validate --file nginx-template.yml
Validating the template file [/Users/james/nginx-template.yml] ...
OK: Syntax of template file [/Users/james/nginx-template.yml] is ok
```

Now run the command `template create`.

```
$ hammr template create --file nginx-template.yml
Validating the template file [/Users/james/nginx-template.yml] ...
OK: Syntax of template file [/Users/james/nginx-template.yml] is ok
Creating template from temporary [/var/folders/f6/8kljm7cx3h7fvb26tq18kw4m0000gn/T/
↪hammr-15888/archive.tar.gz] archive ...
100%|################################################################################|
OK: Template create: DONE
Template URI: users/root/appliances/898
Template Id : 898
```

This takes the information in the `stack` section of the template configuration file and stores this in the UForge server.

You can display all the templates created by using `template list`.

```
$ hammr template list
+-----+----------------------+---------+--------------------+---------------------+--
↪------------------+--------+---------+-----+--------+
| Id  |         Name         | Version |         OS         |       Created       | ␣
↪  Last modified    | # Imgs | Updates | Imp | Shared |
+=====+======================+=========+====================+=====================+==========
 683 | nginx                | 1.0     | Ubuntu 12.04 x86_64 | 2014-05-02 13:59:25 |␣
↪2014-05-02 13:59:27 | 0      | 0       |     |        |
+-----+----------------------+---------+--------------------+---------------------+--
↪------------------+--------+---------+-----+--------+
Found 1 templates
```

You can create one or more machine images from this template.

Building a Machine Image

Once a template has been created, you can create a machine image from it. You can build as many machine images as you like for different platforms and environments. The result will be near identical machine images every time you build from the template. There will be minor differences depending upon the target platform. For example, building an Amazon EC2 image will automatically include the mandatory Amazon libraries required by EC2 to correctly provision an instance, while for OpenStack or VMware vCloud Director, these libraries are not required.

To build a machine image, you need to add the `builders` section to your file. The `builder` section provides mandatory parameters to build (and for some environments register) the machine image. Each target environment requires different `builders` parameters. Refer to the documentation for more information.

For security reasons, it is recommended not to add any cloud account information into the template file. Hammr provides various mechanisms to provide this cloud account information. The method we will use in this tutorial will be to register the cloud account information to the UForge server, then reference the cloud account tag name in the template. So create a file `aws-account.yml` (or `aws-account.json` if you are using JSON) and add the following content:

```
---
accounts:
- type: Amazon
  name: James AWS Account
  accountNumber: 11111-111111-1111
  accessKeyId: myaccessKeyid
  secretAccessKeyId: mysecretaccesskeyid
  x509Cert: "/home/developer/UShareSoft/WKS/Hammr/tests/certs/aws/aws.cert.pem"
  x509PrivateKey: "/home/developer/UShareSoft/WKS/Hammr/tests/certs/aws/aws.key.pem"
```

If you are using JSON:

```
{
  "accounts": [
    {
      "type": "Amazon",
      "name": "James AWS Account",
      "accountNumber": "11111-111111-1111",
      "accessKeyId": "myaccessKeyid",
      "secretAccessKeyId": "mysecretaccesskeyid",
      "x509Cert": "/home/developer/UShareSoft/WKS/Hammr/tests/certs/aws/aws.cert.pem",
      "x509PrivateKey": "/home/developer/UShareSoft/WKS/Hammr/tests/certs/aws/aws.key.
↪pem"
    }
```

```
  ]
}
```

To create the cloud account, use the command `account create`, where `--file` is the YAML or JSON file you created.

```
$ hammr account create --file aws-account.yml
Validating the template file [aws-account.yml] ...
OK: Syntax of template file [aws-account.yml] is ok
Create account for 'ami'...
OK: Account create successfully for [ami]
```

Once the cloud account is created, we can safely reference the cloud credentials in all the template files by using the account name, in this example: `James AWS Account`

Lets now use this account to build a machine image for Amazon EC2. Open up the file `nginx-template.yml`, and provide the following content:

```
---
stack:
  name: nginx
  version: '1.0'
  os:
    name: Ubuntu
    version: '12.04'
    arch: x86_64
    profile: Minimal
    pkgs:
    - name: iotop
  installation:
    diskSize: 12288
builders:
- type: Amazon
  account:
    name: James AWS Account
  installation:
    diskSize: 10240
  region: eu-west-1
  s3bucket: mybucketname
```

If you are using JSON (file `nginx-template.json`):

```
{
  "stack": {
    "name": "nginx",
    "version": "1.0",
    "os": {
      "name": "Ubuntu",
      "version": "12.04",
      "arch": "x86_64",
      "profile": "Minimal",
      "pkgs": [
        {
          "name": "iotop"
        }
      ]
    },
    "installation": {
```

```
      "diskSize": 12288
    }
  },
  "builders": [
    {
      "type": "Amazon",
      "account": {
        "name": "James AWS Account"
      },
      "installation": {
        "diskSize": 10240
      },
      "region": "eu-west-1",
      "s3bucket": "mybucketname"
    }
  ]
}
```

You will notice that the new `builders` section includes the `account` name created earlier as well as the `region` and `bucket` where you will register the machine image.

To build the machine image, use the command `template build`.

```
$ hammr template build --file nginx-template.yml
Validating the template file [nginx-template.yml] ...
OK: Syntax of template file [nginx-template.yml] is ok
Generating 'ami' image (1/1)
|>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| 100%: Done, created on ... |<<<<<<<<<<<<<<<<<<<<<<<<
↪<<<<<<<<<<|
OK: Generation 'ami' ok
Image URI: users/root/appliances/21/images/47
Image Id : 47
```

Publishing and Registering the Machine Image

Once the machine image is generated, you can upload and register the machine image to the target environment, in this case AWS.

> **Warning:** The image uploaded will be stored in AWS S3 storage. AWS does not charge you for any inbound data, however they will charge you for the storage used.

To get the id of the machine image generated, use the command `image list`

```
$ hammr image list
Getting all images and publications for [root] ...
Images:
+------+--------------+---------+------+-----------+---------------------+------+----
↪--------+------------------+
|  Id  |     Name     | Version | Rev. |  Format   |       Created       | Size |␣
↪Compressed | Generation Status |
+======+==============+=========+======+===========+=====================+======+============+======
| 1042 | generation   | 1.0     | 1    | kvm       | 2014-05-21 09:29:36 | 0B   | X ␣
↪        | Done              |
+------+--------------+---------+------+-----------+---------------------+------+----
↪--------+------------------+
```

```
| 981  | wordpress     | 1.0     | 1    | vbox      | 2014-05-19 17:08:06 | 0B   | X ␣
↪        | Canceled          |
+------+---------------+---------+------+-----------+---------------------+------+----
↪--------+-------------------+
| 960  | nginx-muppets | 1.0     | 1    | vbox      | 2014-05-15 13:33:43 | 0B   | X ␣
↪        | Done              |
+------+---------------+---------+------+-----------+---------------------+------+----
↪--------+-------------------+

Found 3 images
No publication available
```

Now use the command `publish` to start the registration process. The following example assumes you are using YAML but you can also use JSON.

```
$ hammr image publish --id <the id> --file nginx-template.yml
Validating the template file [/tmp/centOS.yml] ...
OK: Syntax of template file [/tmp/centOS.yml] is ok
Publishing 'ami' image (1/1)
|>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| 100%: Done, published o... |<<<<<<<<<<<<<<<<<<<<<<
↪<<<<<<<<<<<<<|
OK: Publication to 'amazon' is ok
Cloud ID : ami-25aa6752
```

Next Steps

And thats it! You have just built and published your first machine image with hammr. You should now have a grasp of the basics of hammr; how to create a template; use it to generate a machine image and publish it to the target environment ready for provisioning.

You should now be ready to start using hammr in real world scenarios. For more detailed information of hammr and the features it provides, please use the documentation section.

Any questions or comments, please get in touch by using the mailing list. If you find any bugs, please post them on the issue tracker in GitHub.

CHAPTER 3

Authentication

Communication between hammr and the UForge server is done via HTTPS. To send requests to the UForge server, hammr needs the following information:

- UForge Server URL endpoint
- Your account user name
- Your password

This information can be passed to hammr either from command-line options or from a file.

Command-line Parameters

Authentication information can be passed to hammr via command-line options. These options are:

- `-a` or `--url`: the UForge Server URL endpoint. If the URL uses HTTPS, then the connection will be done securely (recommended), otherwise connection will be done via HTTP
- `-u` or `--user`: the user name to use for authentication
- `-p` or `--password`: the password to use for authentication

For example

```
$ hammr os list --url https://uforge.usharesoft.com/api -u username -p password
```

These parameters need to be passed each time you want to use the command-line.

CHAPTER 4

Using a Credential File

Rather than passing the authentication information as part of the command-line, you can instead store this information in a credential file (`credentials.json` or `credentials.yml`) that will be used every time hammr is launched. Hammr searches for this file in a sub-directory named `.hammr` located in the home directory of the user launching hammr.

For more information, refer to *Using a Credential File*.

CHAPTER 5

Command-Line

Hammr is launched by using the main command-line tool hammr. The hammr tool can be launched with other `commands`, `sub-commands` and `options`. The usage for the tool, can be shown by running hammr with the help options `-h` or `--help`:

```
$ hammr --help
usage: hammr [-h] [-a URL] [-u USER] [-p PASSWORD] [-v] [cmds [cmds ...]]
```

The global `options` are:

- `-h`, `--help`: displays the usage
- `-a URL`, `--url URL`: the UForge server URL endpoint to use
- `-u USER`, `--user USER`: the user name used to authenticate to the UForge server
- `-p PASSWORD`, `--password PASSWORD`: the password used to authenticate to the UForge server
- `-v`: displays the current version of the hammr tool

Hammr communicates with a UForge server instance, requiring authentication information as part of this authentication. The authentication information may be passed to hammr via the global options, however, there are other ways to pass this information. For more information, please refer to *Authentication* section of the documentation.

Below provides a list of the available `commands`:

account

Manages all of your different cloud accounts used when either building or publishing machine images. The usage is:

```
usage: hammr account [sub-command] [options]
```

Sub Commands

`create` sub-command

Creates a new cloud account. The options are:

- `--file` (mandatory): json or yaml file providing the cloud account parameters

`delete` sub-command

Deletes an existing cloud account. The options are:

- `--id` (mandatory): the ID of the cloud account to delete

`list` sub-command

Displays all the cloud accounts for the user.

bundle

Manages all the bundles that have been registered in UForge. A bundle is group of software that is uploaded during the creation of a template. The usage is:

```
usage: hammr bundle [sub-command] [options]
```

Sub Commands

`categories` sub-command

Lists all the categories available for bundles.

`clone` sub-command

Clones the bundle. The clone is copying the meta-data of the bundle. The options are:

- `--id` (mandatory): the ID of the bundle to clone
- `--name` (mandatory): the name to use for the new cloned bundle
- `--version` (mandatory): the version to use for the cloned bundle

`create` sub-command

Creates a new bundle and saves it to the UForge server. Hammr creates a tar.gz archive which includes the .json or .yml file and binaries and imports it to UForge. The options are:

- `--file` (mandatory): json or yaml file containing the bundle content. See the *files* sub-section for available keys.
- `--archive-path` (optional): path of where to store the archive (tar.gz) of the created bundle. If provided, hammr creates an archive of the created bundle, equivalent to running `bundle export`

`delete` sub-command

Deletes an existing bundle. The options are:

- `--id` (mandatory): the ID of the bundle to delete

`export` sub-command

Exports a bundle by creating an archive (compressed tar file) that includes the .json or .yml bundle configuration file. The options are:

- `--id` (mandatory): the ID of the bundle to export
- `--file` (optional): destination path where to store the bundle configuration file on the local filesystem
- `--outputFormat` (optional): output format (yaml or json) of the bundle file to export (yaml is the default one)

`import` sub-command

Creates a bundle from an archive. The archive file must be a tar.gz (which includes the .json or .yml and binaries). The options are:

- `--file` (mandatory): the path of the archive

`list` sub-command

Lists all the bundles that have been registered in the UForge server.

`validate` sub-command

Validates the syntax of a bundle configuration file. The options are:

- `--file` (mandatory): the json or yaml configuration file

format

Displays all the machine image formats the user has access to when building a machine image. The usage is:

```
usage: hammr format [sub-command]
```

Sub Commands

`list` sub-command

Displays all the machine image formats for the user.

platform

Displays all the platform types the user has access to for creating cloud accounts. The usage is:

```
usage: hammr platform [sub-command]
```

Sub Commands

`list` sub-command

Displays all the platform types available for the user.

image

Manages all of the machine images you have built and/or published. The usage is:

```
usage: hammr image [sub-command] [options]
```

Sub Commands

`cancel` sub-command

Cancels a machine image build or publish. The options are:

- `--id` (mandatory): the ID of the machine image to cancel

`delete` sub-command

Deletes a machine image or publish information. The options are:

- `--id` (mandatory): the ID of the machine image to delete

`download` sub-command

Downloads a machine image to the local filesystem. The options are:

- `--id` (mandatory): the ID of the machine image to download
- `--file` (mandatory): the pathname where to store the machine image

`list` sub-command

Displays all the machine images built and publish information of those machine images to their respective target platforms.

`publish` sub-command

Publish (upload and register) a built machine image to a target environment. The options are:

- `--file` (mandatory): json or yaml file providing the cloud account parameters required for upload and registration

os

Displays all the operating systems available to use when creating templates. The usage is:

```
usage: hammr os [sub-command] [options]
```

Sub Commands

`list` sub-command

Displays all the operating systems available to use by the user.

`search` sub-command

Operating System package search.

- `--id` (mandatory): the ID of the OS
- **`--pkg` (mandatory): Regular expression of the package:**
  - "string" : search all packages wich contains "string"
  - "^string": search all packages wich start with "string"
  - "string$": search all packages wich end with "string"

quota

Displays the current user quota for the user. This includes all of your different cloud accounts used when either building or publishing machine images. The usage is:

```
usage: hammr quota [sub-command]
```

Sub Commands

`list` sub-command

Displays the user's quota information.

scan

Manages all the scans executed on live systems. The usage is:

```
usage: hammr scan [sub-command] [options]
```

Sub Commands

`build` sub-command

Builds a machine image from a scan. The options are:

- `--id` (mandatory): the ID of the scan to generate the machine image from
- `--file` (mandatory): json or yaml file providing the builder parameters

---

Note: When building from a scan, your yaml or json file must contain an `installation` and `hardwareSettings` section in `builders`. Refer to *installation* for installation details and *Builders* for the hardware settings, which depend on the builder type.

`delete` sub-command

Deletes an existing scan. The options are:

- `--id` (mandatory): the ID of the instance or scan to delete
- `--scantype` (mandatory): the type to be deleted. Can be one of: instance, scan, or all. When you set the type to `instance`, the instance and all scans linked to it will be deleted unless using the `scansonly` flag. When you specify the type as `scan` only the scan with the ID to specify will be deleted. If you set the type to `all`, all the instances and scans on your UForge will be deleted (regardeless of the `id` you set).
- `--scansonly` (optional): this flag can be used when the scan type is set to `instance`. In this case, only the scans linked to the specified instance will be deleted (not the instance itself).

`import` sub-command

Imports (or transforms) the scan to a template.

- `--id` (mandatory): the ID of the scan to import
- `--name` (mandatory): the name to use for the template created from the scan
- `--version` (mandatory): the version to use for the template created from the scan

`list` sub-command

Displays all the scans for the user.

`run` sub-command

Executes a deep scan of a running system.

- `--ip` (mandatory): the IP address or fully qualified hostname of the running system
- `--scan-login` (mandatory): the root user name (normally root)
- `--name` (mandatory): the scan name to use when creating the scan meta-data
- `--scan-password` (optional): the root password to authenticate to the running system
- `--dir` (optional): the directory where to install the uforge-scan.bin binary used to execute the deep scan
- `--exclude` (optional): a list of directories or files to exclude during the deep scan
- `--overlay` (optional): include overlay (extra files) for the given scan

template

Manages all the templates created by the user. The usage is:

```
usage: hammr template [sub-command] [options]
```

Sub Commands

`build` sub-command

Builds a machine image from the template. The options are:

- `--file` (mandatory): json or yaml file providing the builder parameters

`clone` sub-command

Clones the template. The clone is copying the meta-data of the template. The options are:

- `--id` (mandatory): the ID of the template to clone
- `--name` (mandatory): the name to use for the new cloned template
- `--version` (mandatory): the version to use for the cloned template

`create` sub-command

Creates a new template and saves it to the UForge server. Hammr creates a tar.gz archive which includes the JSON or YAML file and binaries, and imports it to UForge. The options are:

- `--file` (mandatory): json or yaml file containing the template content
- `--archive-path` (optional): path of where to store the archive (tar.gz) of the created template. If provided, hammr creates an archive of the created template, equivalent to running `template export`
- `--force` (optional): force template creation (delete template/bundle if already exist)
- `--rbundles` (optional): if a bundle already exists, use it in the new template. Warning: this option ignore the content of the bundle described in the template file
- `--usemajor` (optional): use distribution major version if exit

`delete` sub-command

Deletes an existing template. The options are:

- `--id` (mandatory): the ID of the template to delete

`export` sub-command

Exports a template by creating an archive (compressed tar file) that includes the JSON or YAML configuration file. The options are:

- `--id` (mandatory): the ID of the template to export
- `--file` (optional): destination path where to store the template configuration file on the local filesystem
- `--outputFormat` (optional): output format (yaml or json) of the template file to export (yaml is the default one)

`import` sub-command

Creates a template from an archive. The archive file must be a tar.gz (which includes the .json or yaml, and binaries). The options are:

- `--file` (mandatory): the path of the archive
- `--force` (optional): force template creation (delete template/bundle if already exist)
- `--usemajor` (optional): use distribution major version if exit

`list` sub-command

Displays all the created templates.

`validate` sub-command

Validates the syntax of a template configuration file. The options are:

- `--file` (mandatory): the json or yaml configuration file

CHAPTER 6

Your Account

Since hammr allows you to communicate with and use the UForge server, a number of actions are managed by your UForge Account, such as the operating systems you have access to, any quotas set (on the number of images you can create, for example), as well as managing your cloud account. The following sections will help you use hammr to access information from your UForge cloud account.

Operating Systems

Hammr allows you to create machine images for a number of OSes. The type of OS you want to use needs to be defined in the `os` section of the configuration file, as described in.

For a list of the OSes that can be added to your template, run `os list`, for example

```
$ hammr os list
Getting distributions for [root] ...
+-----+---------+--------------+--------------+---------------------+-----------------
↪---------+
| Id  |  Name   |   Version    | Architecture |    Release Date     |        ␣
↪Profiles         |
+=====+=========+==============+==============+=====================+=========================+
| 120 | CentOS  | 6            | x86_64       | 2011-07-03 02:06:43 | Server          ␣
↪         |
|     |         |              |              |                     | Minimal         ␣
↪         |
|     |         |              |              |                     | Minimal Desktop␣
↪         |
+-----+---------+--------------+--------------+---------------------+-----------------
↪---------+
| 121 | CentOS  | 6            | i386         | 2011-07-03 04:02:09 | Minimal         ␣
↪         |
|     |         |              |              |                     | Server          ␣
↪         |
|     |         |              |              |                     | Minimal Desktop␣
↪         |
```

```
+-----+---------+-------------+-------------+--------------------+-----------------
↪---------+
| 122 | CentOS  | 5           | x86_64      | 2008-06-19 13:56:25 | Minimal Desktop␣
↪         |
|     |         |             |             |                     | Minimal        ␣
↪         |
|     |         |             |             |                     | Server         ␣
↪         |
+-----+---------+-------------+-------------+--------------------+-----------------
↪---------+
| 123 | CentOS  | 5           | i386        | 2008-06-19 14:01:21 | Minimal        ␣
↪         |
|     |         |             |             |                     | Minimal Desktop␣
↪         |
|     |         |             |             |                     | Server         ␣
↪         |
+-----+---------+-------------+-------------+--------------------+-----------------
↪---------+
| 42  | Debian  | 6           | x86_64      | 2010-04-20 00:18:34 | Minimal Desktop␣
↪         |
|     |         |             |             |                     | Server         ␣
↪         |
|     |         |             |             |                     | Minimal        ␣
↪         |
+-----+---------+-------------+-------------+--------------------+-----------------
↪---------+
| 125 | Debian  | 7           | x86_64      | 2012-11-05 11:17:46 | Minimal Desktop␣
↪         |
|     |         |             |             |                     | Server         ␣
↪         |
|     |         |             |             |                     | Minimal        ␣
↪         |
+-----+---------+-------------+-------------+--------------------+-----------------
↪---------+
| 124 | Debian  | 7           | i386        | 2012-11-05 11:17:46 | Server         ␣
↪         |
|     |         |             |             |                     | Minimal        ␣
↪         |
|     |         |             |             |                     | Minimal Desktop␣
↪         |
+-----+---------+-------------+-------------+--------------------+-----------------
↪---------+
Found 7 distributions
```

Quotas

There are a number of quotas that can be set on a UForge account. For example, a free account has the following limitations:

Quotas can be set for the following:

- Disk usage: diskusage in bytes (includes storage of bundle uploads, bootscripts, image generations, scans)
- Templates: number of templates created
- Generations: number of machine images generated

- Scans: number of scans for migration

To view the quotas that have been set on your account, run `quota list`:

```
$ hammr quota list
Getting quotas for [root] ...
Scans (25)                 --------------------UNLIMITED---------------------
Templates (26)             --------------------UNLIMITED---------------------
Generations (72/100)       ||||||||||||||||||||||||||||||||||||--------------
Disk usage (30GB)          --------------------UNLIMITED---------------------
```

The output not only lists any quotas that are set, but it also shows you the limit you are at, even if your account is set to unlimited.

Setting Your Cloud Accounts

For security reasons, it is recommended not to add any cloud account information into the template file. Hammr allows you to register your cloud account information to the UForge server, then reference the cloud account tag name in the template.

To do this, you need to create a JSON or YAML file which contains all the necessary cloud credentials. This will depend on your cloud type. For more information, refer to the *Builders* section of the documentation.

Once this file is ready, you create the cloud account on UForge by running the command `account create`. The following example assumes you have created a YAML file, but you can also use JSON.

```
$ hammr account create --file aws-account.yml
Validating the template file [aws-account.yml] ...
OK: Syntax of template file [aws-account.yml] is ok
Create account for 'ami'...
OK: Account create successfully for [ami]
```

Once the cloud account is created, you can safely reference the cloud credentials in all the template files by using the account name.

CHAPTER 7

Creating and Managing Templates

If you have read the Introduction and gone through the step by step tutorial in the Getting Started section, you now have an operational configuration template and have built an image. However, the example probably doesn't match with the type of machine image you want to create. This section will help you with the basics for creating and managing your template, including how to manage package updates.

You should refer to the section for all the possible parameters you can add to define your template.

Creating a Template

A template is a configuration file which defines the machine image you want to build. The format of this file is either JSON or YAML. Note that the template can be saved locally or stored on a server, in which case hammr will access it via a URL. For security reasons we recommend that you save your UForge credentials in a seperate credentials file, saved at the same location as your template.

The mandatory values when creating a template are:

- `name`: the name of the template to create. You can easily make the name unique by using the timestamp keyword (surrounded by curly brackets).
- `version`: the version of the template.
- `os` : the operating system details to use the in your images. You must include the OS family name, version and architecture type. For more information regarding OS and package parameters, see *Adding Packages to Your Template*

For more details about the various parameters you can set in your template to define the machine image you want to create, refer to the *Templates Specification* section.

The following is an example of the minimum information needed in your configuration file to define a template. It includes the name, version, a few installation parameters and the OS. The following YAML example describes a CentOS 6.4 32-bit template. JSON can also be used.

```
---
stack:
```

```
  name: myTemplate
  version: '1.0'
  installation:
    internetSettings: basic
    diskSize: 12288
    swapSize: 512
  os:
    name: CentOS
    version: '6.4'
    arch: x86_64
    profile: Minimal
```

If you are using JSON:

```
{
  "stack" : {
    "name" : "myTemplate",
    "version" : "1.0",
    "installation" : {
      "internetSettings" : "dhcp",
      "diskSize" : 12288,
      "swapSize" : 512
    },
    "os" : {
      "name" : "CentOS",
      "version" : "6.4",
      "arch" : "x86_64",
      "profile" : "Minimal"
    }
 }
}
```

Once you have written and saved the minimal template you can then create the template using `template create`:

```
$ hammr template create --file <blueprint>.yml
Validating the template file [/Users/james/nginx-template.yml] ...
OK: Syntax of template file [/Users/james/nginx-template.yml] is ok
Creating template from temporary [/var/folders/f6/8kljm7cx3h7fvb26tq18kw4m0000gn/T/
↪hammr-15888/archive.tar.gz] archive ...
100%|################################################################################|
OK: Template create: DONE
Template URI: users/root/appliances/898
Template Id : 898
```

Validating Your Template

Once you have created and modified your template file, it is best practice to validate your template before you build or publish it. In order to check that your template does not have any syntax errors or missing mandatory values, run the command `validate`. The following example assumes you are using a YAML file but you can also use JSON.

```
$ hammr template validate --file <path/filename>.yml
Validating the template file [/Users/james/nginx-template.yml] ...
OK: Syntax of template file [/Users/james/nginx-template.yml] is ok
```

If there are any errors, this command will tell you.

Adding Packages to Your Template

When defining your machine image you set the OS and profile. UForge automatically pulls in all the necessary packages required for the chosen OS. You do not need to list them seperately. However, you may want to add other packages to your machine image. These additional packages are listed in the `pkgs` section of your template.

Note: If the packages you choose to add to your template have any dependencies, all the required packages will be added automatically. You do not have to search and list all the dependencies in your template.

The following is a basic example for a CentOS 6.4 32-bit template with package for `iotop` added, when using YAML.

```
---
os:
  name: CentOS
  version: '6.4'
  arch: x86_64
  profile: Minimal
  pkgs:
        name: iotop
```

If you are using JSON:

```
{
    "os" : {
      "name" : "CentOS",
      "version" : "6.4",
      "arch" : "x86_64",
      "profile" : "Minimal"
      "pkgs" : {
        "name" : "iotop"
      }
    }
}
```

Searching for Packages

You can search for the available packages as follows:

```
$ hammr os search
```

When running a search you will need to specify the OS `id` and a search string.

```
$ hammr os search --id 121 --pkg ntpdate
Search package 'ntpdate' ...
for OS 'CentOS', version 6
+---------+---------+------+--------------+---------------------+------+--------------
↪-----------------------------+
|  Name   | Version | Arch |   Release    |     Build date      | Size |              ␣
↪   FullName                   |
+=========+=========+======+==============+=====================+======+==============================
| ntpdate | 4.2.4p8 | i686 | 3.el6.centos | 2013-02-22 11:22:14 | 56K  |   ntpdate-4.
↪2.4p8-3.el6.centos.i686.rpm   |
+---------+---------+------+--------------+---------------------+------+--------------
↪-----------------------------+
```

```
| ntpdate | 4.2.4p8 | i686 | 2.el6.centos | 2011-11-29 12:06:40 | 56K  |   ntpdate-4.
↪2.4p8-2.el6.centos.i686.rpm   |
+---------+---------+------+--------------+---------------------+------+--------------
↪-----------------------------+
| ntpdate | 4.2.4p8 | i686 | 2.el6        | 2010-08-25 01:51:27 | 56K  |   ntpdate-4.
↪2.4p8-2.el6.i686.rpm          |
+---------+---------+------+--------------+---------------------+------+--------------
↪-----------------------------+
| ntpdate | 4.2.6p5 | i686 | 1.el6.centos | 2013-11-23 06:20:19 | 74K  |   ntpdate-4.
↪2.6p5-1.el6.centos.i686.rpm   |
+---------+---------+------+--------------+---------------------+------+--------------
↪-----------------------------+

Found 4 packages
```

To get the OS `id`, list the OS information by running:

```
$ hammr os list
```

Understanding Package Updates

A more complete example for adding CentOS is provided below. You will notice the following optional information has been added:

- `updateTo`: This is the date up until which the packages should be updated
- `profile`: The OS profile. The options are listed under os list

If you are using YAML:

```
---
os:
  name: CentOS
  version: '6.4'
  arch: x86_64
  updateTo: 01-30-2014
  profile: Minimal
  pkgs:
  - name: iotop
  - name: httpd
    version: 2.2.15
    release: 28.el6.centos
    arch: x86_64
```

If you are using JSON:

```
{
  "os" : {
    "name" : "CentOS",
    "version" : "6.4",
    "arch" : "x86_64",
    "updateTo" : "01-30-2014",
    "profile" : "Minimal",
    "pkgs" : [ {
      "name" : "iotop"
    },
```

```
    {
      "name" : "httpd",
      "version" : "2.2.15",
      "release" : "28.el6.centos",
      "arch" : "x86_64"
    }]
  }
}
```

In the example above you can see that for the package httpd a specific version and release are specified. When no version or release is specified, the latest release is used.

Package Dependencies and Updates

Hammr, via the build mechanism, determines the complete list of packages that must be installed by checking the dependencies of the packages you have listed in the `os` sub-section of the stack (via the profile and pkgs) and any native packages listed in a `bundle`. Hammr also provides a mechanism to track any available updates on these packages and allows you to update or roll-back your template.

What is a Dependency

A dependency is a piece of information in a software package that describes which other packages it requires to function correctly. Many packages require operating system libraries as they provide common services that just about every program uses (filesystem, network, memory etc).

For example, network applications typically depend on lower-level networking libraries provided by the operating system. The principle behind package dependencies is to share software, allowing software developers to write and maintain less code at a higher quality. Operating systems have thousands of packages.

In the world of virtualization and cloud computing, it is becoming imperative to strip down the number of operating system packages to just the required packages to run a particular application. This process, known as JeOS (pronounced "juice") standing for "Just Enough Operating System" is a very painful manual process. So much so that many operating system vendors now supply a core operating system ISO with the minimum set of packages required to boot the system. The fun then begins as you manually install only the packages (and their dependencies) required to run your application.

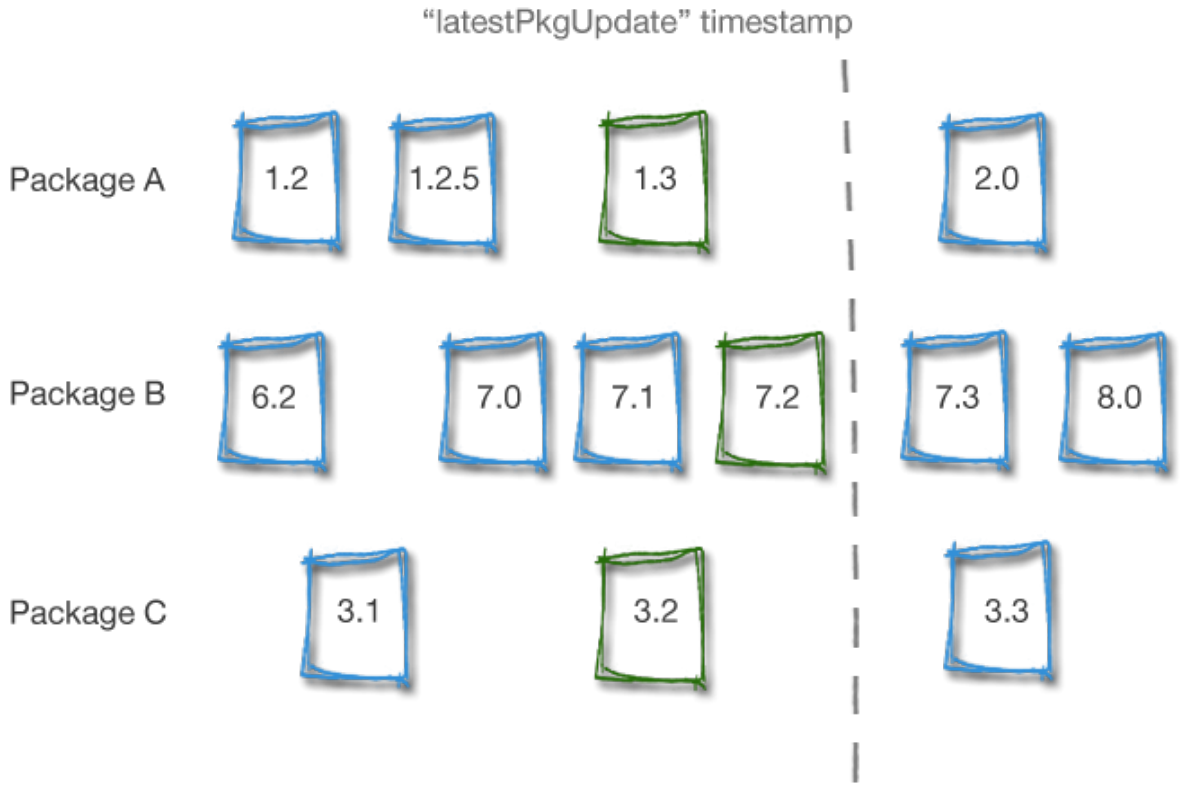
Calculating Package Dependencies

Package dependency checking occurs when you build (or generate) a new machine image. During the first phase of generation, the backend UForge Server calculates automatically all the dependencies of each package in the os section (profile and pkgs list) as well as any packages contained elsewhere in your stack (native packages declared in one or more bundles).

All missing packages are automatically added. For each package added, this package's dependencies are also checked. This process continues until all the dependencies have been met. The end result is a complete dependency tree of all the packages you require to run your application. All these packages are added to the machine image. Consequently you should not be surprised if the number of packages that are actually installed are larger than the packages listed in the stack section of the template.

Each package has meta-data on what the package requires (that is, what the package depends on) and what it provides in terms of functionality. This meta-data varies on the package type (RPM, DEB etc).

The dependency calculation is done using a specific moment in time. This date is determined by the `updateTo` key in your stack. If this key does not exist, then the date the template was created (via the command `template create`) is used. Chosen package versions and dependencies are calculated by ensuring that they are equal to or less than this date. Let's take an example. Imagine you create a new stack on June 17th 2013, 17:00 GMT+1, and you choose package A, B and C. Packages A, B and C may have more than one version (updates added to the repository due to bug fixing and or new features). The versions displayed for A, B and C will be dates of each of these packages closest (but inferior) to our date.



Package Updates

As you probably know, packages evolve as bugs are fixed and new features are added. These new packages become available in the operating system repository. The UForge Server uses an internal mechanism to check for any new package update available in the repository, and, if found, adds the meta-data of this package to its own database. Using this process, the UForge Server builds a history of the operating system, as it keeps references to the old packages that are being replaced by the update.

These updates do not get taken into account for your current template when generating a new image. By calculating packages by the same date ensures when you build your machine image, the same image is generated time after time. This is due to always using the `updateTo` date (or the creation date) of the template in question.

Ok great, but what if you actually wanted to include these updates in the next generation? Well, it's a simple matter of updating the `updateTo` key of the stack section.

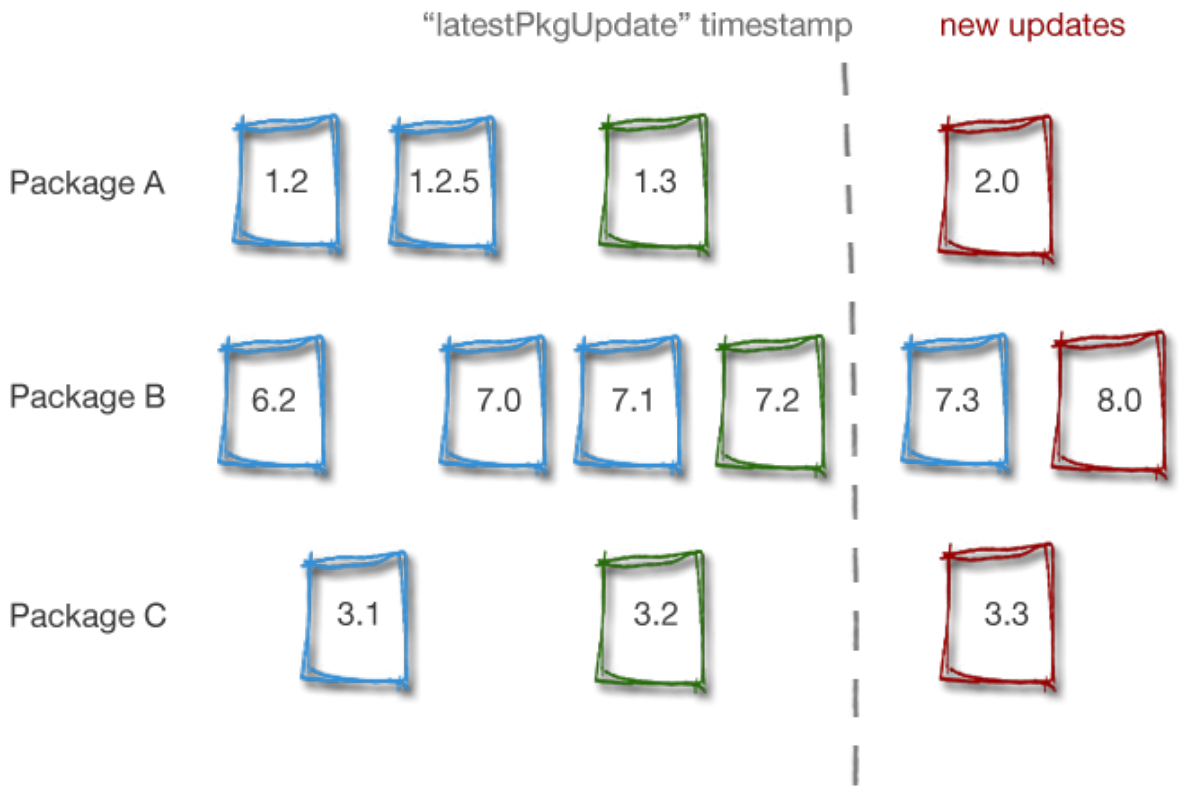
If you are using YAML:

```
---
stack:
  name: CentOS Base Template
```

```
  version: '6.4'
  description: This is a CentOS core template.
  os:
        name: CentOS
        version: '6.4'
        arch: x86_64
        profile: Minimal
        updateTo: '2013-06-15'
```

If you are using JSON:

```
{
        "stack": {
            "name": "CentOS Base Template",
            "version": "6.4",
            "description": "This is a CentOS core template.",
            "os": {
              "name": "CentOS",
              "version": "6.4",
              "arch": "x86_64",
              "profile": "Minimal",
              "updateTo": "2013-06-15"
            }
        }
}
```



In this case, UForge will notify you that three updates are available. Note, that for package B even if there is an intermediary package (version 7.3), only the last one is taken into account.

Pinning a Package

Being able to roll-forward or roll-back the packages is all well and good, but what if we wanted to force a particular version of a package to be part of the machine image?

Due to the current package version calculation being based on a particular date it is impossible to specify a particular package version to be part of the generation, as depending upon the build date of the package, potentially an earlier or more up to date version of the package may be chosen instead. To get around this issue, hammr provides a mechanism to force a particular package version. This is known as "pinning" a package (previously referred to as macking a package "sticky"). To do this, specify the fullname of the package, or its version, revision and architecture.

For example when using YAML:

```
---
stack:
  name: CentOS Base Template
  version: '6.4'
  description: This is a CentOS core template.
  os:
    name: CentOS
    version: '6.4'
    arch: x86_64
    profile: Minimal
    updateTo: '2013-06-15'
    pkgs:
    - name: php
      version: 5.5.3
      release: 23.el6_4
      arch: i686
    - name: php-common
      fullName: php-common-5.5.3-23.el6_4-i686.rpm
```

If you are using JSON:

```
{
  "stack": {
    "name": "CentOS Base Template",
    "version": "6.4",
    "description": "This is a CentOS core template.",
    "os": {
      "name": "CentOS",
      "version": "6.4",
      "arch": "x86_64",
      "profile": "Minimal",
      "updateTo": "2013-06-15",
      "pkgs": [
        {
          "name": "php",
          "version": "5.5.3",
          "release": "23.el6_4",
          "arch": "i686"
        },
        {
          "name": "php-common",
          "fullName": "php-common-5.5.3-23.el6_4-i686.rpm"
        }
      ]
    }
```

```
  }
}
```

Using Advanced Partitioning

Hammr supports the ability to describe partitioning schemas as part of the stack used to build machine images. Partitioning is the act of dividing one or more physical disks into logical sections with the goal to treat each physical disk drive as if it were multiple disks.

Partitioning is used frequently in production systems. Benefits include:

- Isolating data from programs
- Keeping frequently used programs and data near each other
- Having cache and log files separate from other files. These can change size dynamically and rapidly, potentially making a file system full.
- Having a separate area for operating system virtual memory swapping/paging

Warning: Some cloud platforms do not support all the features of partitioning, or limit the number of partitions you may have in your machine image. When building a machine image for a particular cloud platform, hammr will return an error if the partitioning setup is not supported. This helps you save time and effort down the road when an instance of the machine image does not boot.

The rest of this section provides examples of:

Disks

The first thing a partitioning table needs is to declare one or more disks that will be used to partition. Each disk declared in the partitioning table has the name `sd` followed by a letter, starting at `a`, namely: 1st disk `sda`, 2nd disk `sdb` and so on. A disk is one of two types, either `MSDOS` or `LVM`, and provides a total disk size available. LVM disks cannot have any physical partitions, however can be used in logical volumes (refer to *Volume Groups and Logical Volumes*).

The example below describes 1 disk of 20GB when using YAML.

```
---
installation:
  partitioning:
        disks:
        - name: sda
          type: msdos
          size: 20480
```

If you are using JSON:

```
{
  "installation": {
    "partitioning": {
      "disks": [
        {
          "name": "sda",
```

```
          "type": "msdos",
          "size": 20480
        }
      ]
    }
  }
}
```

Example

The following example describes 2 disks of 20GB each when using YAML.

```
---
installation:
  partitioning:
        disks:
        - name: sda
          type: msdos
          size: 20480
        - name: sdb
          type: msdos
          size: 20480
```

If you are using JSON:

```
{
  "installation": {
    "partitioning": {
      "disks": [
        {
          "name": "sda",
          "type": "msdos",
          "size": 20480
        },
        {
          "name": "sdb",
          "type": "msdos",
          "size": 20480
        }
      ]
    }
  }
}
```

Physical Partitions

Each disk declared may be partitioned, i.e. the act of dividing the physical disk into logical sections with the goal to treat one physical disk drive as if it were multiple disks. These are called physical partitions.

Note: A disk may have a maximum of 4 physical partitions.

Each physical partition has a unique number (1,2,3 and 4) and declare a filesystem type and size. All filesystem types with the exception of `lvm2`, `extended` and `linux-swap` require a mount point. LVM physical partitions are used

in logical volumes (which will be covered later).

Example

The following example shows 3 physical partitions of a disk: `/boot`, `swap`, and `/space`.


```
---
partitioning:
  disks:
  - name: sda
        type: msdos
        size: 20480
        partitions:
        - number: 1
          fstype: ext3
          size: 2048
          mountPoint: "/boot"
        - number: 2
          fstype: linux-swap
          size: 1024
        - number: 3
          fstype: ext3
          size: 17408
          label: space
          mountPoint: "/space"
```

If you are using JSON:

```
{
        "partitioning": {
            "disks": [
              {
                "name": "sda",
                "type": "msdos",
                "size": 20480,
                "partitions": [
                  {
                    "number": 1,
                    "fstype": "ext3",
                    "size": 2048,
                    "mountPoint": "/boot"
                  },
                  {
                    "number": 2,
                    "fstype": "linux-swap",
                    "size": 1024
                  },
                  {
                    "number": 3,
                    "fstype": "ext3",
```

```
                    "size": 17408,
                    "label": "space",
                    "mountPoint": "/space"
                }
            ]
        }
    ]
  }
}
```

Note: In a partitioning table, at least one partition must be the `/boot` partition. In the above example this is one of the physical partitions. Furthermore, the sum of the physical partition sizes must be smaller or equal to the disk size.

Logical Partitions

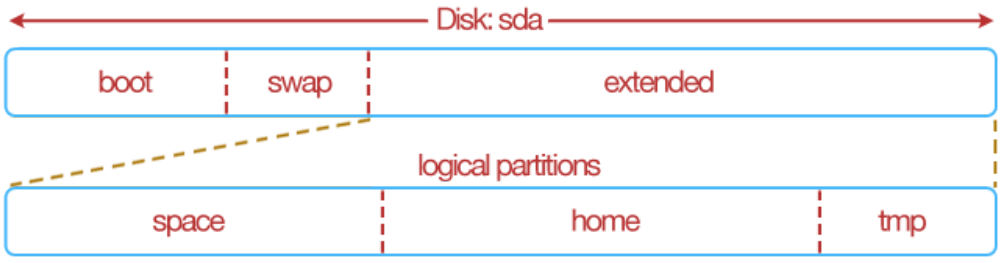
Due to the restriction of only having 4 physical partitions for disk, you can further partition a physical partition using logical partitioning. To partition a physical partition, use the filesystem type `extended`.

Note: You can only use the `extended` filesystem ONCE for a disk i.e. you can only partition one physical partition in a disk. When using extended you cannot declare a mount point or label. These will be ignored by hammr.

Like a physical partition, each logical partition has a unique number starting at 5 (5,6,7,8 etc) and declares a filesystem type and size. You cannot further partition a logical partition (extended filesystem type cannot be used). There is no limit to the number of logical partitions you may have, however the sum of the logical partitions cannot exceed the size of the physical partition.

Example

The following example shows 3 physical partitions of a disk, where the last physical partition has 3 logical partitions : `/space`, `/home` and `/tmp`.




```
---
partitioning:
  disks:
  - name: sda
        type: msdos
        size: 20480
        partitions:
        - number: 1
```

```
          fstype: ext3
          size: 2048
          mountPoint: "/boot"
        - number: 2
          fstype: linux-swap
          size: 1024
        - number: 3
          fstype: Extended
          size: 17408
          partitions:
          - number: 5
                fstype: ext3
                size: 8192
                mountPoint: "/space"
                label: space
          - number: 6
                fstype: ext3
                size: 8192
                mountPoint: "/home"
                label: home
          - number: 7
                fstype: ext3
                size: 1024
                mountPoint: "/tmp"
                label: tmp
```

If you are using JSON:

```
{
        "partitioning": {
            "disks": [
              {
                "name": "sda",
                "type": "msdos",
                "size": 20480,
                "partitions": [
                  {
                    "number": 1,
                    "fstype": "ext3",
                    "size": 2048,
                    "mountPoint": "/boot"
                  },
                  {
                    "number": 2,
                    "fstype": "linux-swap",
                    "size": 1024
                  },
                  {
                    "number": 3,
                    "fstype": "Extended",
                    "size": 17408,
                    "partitions": [
                      {
                        "number": 5,
                        "fstype": "ext3",
                        "size": 8192,
                        "mountPoint": "/space",
                        "label": "space"
```

```
                        },
                        {
                          "number": 6,
                          "fstype": "ext3",
                          "size": 8192,
                          "mountPoint": "/home",
                          "label": "home"
                        },
                        {
                          "number": 7,
                          "fstype": "ext3",
                          "size": 1024,
                          "mountPoint": "/tmp",
                          "label": "tmp"
                        }
                      ]
                    }
                  ]
                }
              ]
          }
}
```

Growable Partitions

Physical and logical partitions can be marked as growable by using the `grow` flag. This declares that the particular partition takes all remaining disk space available after the other partition sizes have been satisfied.

You can only declare one physical partition to be growable in a disk, and one logical partition to be growable for a physical partition.

Example

In this example we mark the "space" physical partition as growable, i.e. the "space" partition takes up the rest of the disk (rather than us having to calculate the space left after creating the first two partitions). We must specify though a size for the "space" partition (the minimum partition size is 64MB).


```
---
partitioning:
  disks:
  - name: sda
        type: msdos
        size: 20480
        partitions:
        - number: 1
          fstype: ext3
          size: 2048
          mountPoint: "/boot"
```

```
        - number: 2
          fstype: linux-swap
          size: 1024
        - number: 3
          fstype: ext3
          size: 64
          grow: true
          label: space
          mountPoint: "/space"
```

If you are using JSON:

```
{
        "partitioning": {
            "disks": [
              {
                "name": "sda",
                "type": "msdos",
                "size": 20480,
                "partitions": [
                  {
                    "number": 1,
                    "fstype": "ext3",
                    "size": 2048,
                    "mountPoint": "/boot"
                  },
                  {
                    "number": 2,
                    "fstype": "linux-swap",
                    "size": 1024
                  },
                  {
                    "number": 3,
                    "fstype": "ext3",
                    "size": 64,
                    "grow": true,
                    "label": "space",
                    "mountPoint": "/space"
                  }
                ]
              }
            ]
    }
}
```

Volume Groups and Logical Volumes

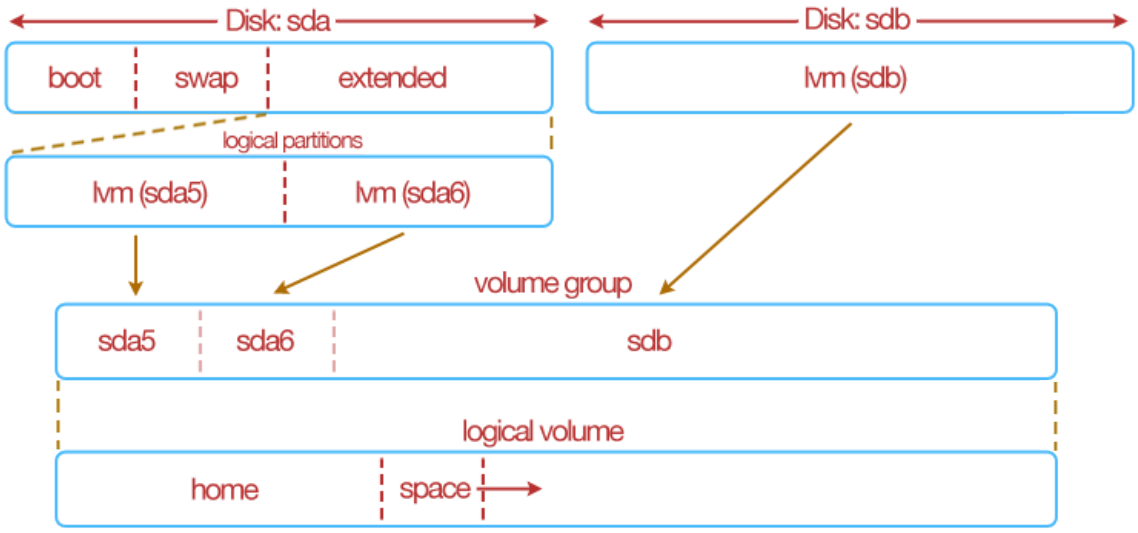
Volume groups and logical volumes allows more creative and flexible partitioning schemas to be created than conventional partitioning schemas we have already discussed.

A volume group allows you to gather disks, physical and logical partitions into a single logical pool of storage. This pool of storage can then be partitioned (like a disk) by using a logical volume.

Only disks that have the type `lvm`, and physical partitions or logical partitions that have filesystem types `lvm2` can be grouped together in a volume group.

Note: Once a physical or logical partition is grouped together into a volume group, they cannot be declared in another volume group.

In this example, an extended physical partition that has two logical partitions that have `lvm` filesystems; and a disk of type `lvm` are pooled together via a volume group `grp1`. A logical volume is used to partition further this volume group.




```
---
partitioning:
  disks:
  - name: sda
        type: msdos
        size: 20480
        partitions:
        - number: 1
          fstype: ext3
          mountPoint: "/boot"
          size: 1024
        - number: 2
          fstype: linux-swap
          size: 1024
        - number: 3
          fstype: extended
          size: 18432
          partitions:
          - number: 5
                fstype: lvm2
                size: 9216
          - number: 6
                fstype: lvm2
                size: 9216
  - name: sdb
        type: lvm
        size: 122880
  volumeGroups:
```

```
  - name: grp1
        physicalVolumes:
        - name: sda5
        - name: sda6
        - name: sdb
  logicalVolumes:
  - name: vol1
        vg_name: grp1
        fstype: ext3
        mountPoint: "/home"
        size: 4098
  - name: vol2
        vg_name: grp1
        fstype: ext3
        mountPoint: "/space"
        size: 64
        grow: true
```

If you are using JSON:

```
{
        "partitioning": {
            "disks": [
              {
                "name": "sda",
                "type": "msdos",
                "size": 20480,
                "partitions": [
                  {
                    "number": 1,
                    "fstype": "ext3",
                    "mountPoint": "/boot",
                    "size": 1024
                  },
                  {
                    "number": 2,
                    "fstype": "linux-swap",
                    "size": 1024
                  },
                  {
                    "number": 3,
                    "fstype": "extended",
                    "size": 18432,
                    "partitions": [
                      {
                        "number": 5,
                        "fstype": "lvm2",
                        "size": 9216
                      },
                      {
                        "number": 6,
                        "fstype": "lvm2",
                        "size": 9216
                      }
                    ]
                  }
                ]
              },
```

```
            {
              "name": "sdb",
              "type": "lvm",
              "size": 122880
            }
          ],
          "volumeGroups": [
            {
              "name": "grp1",
              "physicalVolumes": [
                {
                  "name": "sda5"
                },
                {
                  "name": "sda6"
                },
                {
                  "name": "sdb"
                }
              ]
            }
          ],
          "logicalVolumes": [
            {
              "name": "vol1",
              "vg_name": "grp1",
              "fstype": "ext3",
              "mountPoint": "/home",
              "size": 4098
            },
            {
              "name": "vol2",
              "vg_name": "grp1",
              "fstype": "ext3",
              "mountPoint": "/space",
              "size": 64,
              "grow": true
            }
          ]
        }
}
```

CHAPTER 8

Building and Publishing Machine Images

Templates allow you to generate and publish identical machine images for physical, virtual and cloud environments. The rest of this section highlights how to build, publish and manage machine images with `hammr`.

Building a Machine Image

In order to generate a machine image based on the template you created, you must update the template with the information for each type of image you want to generate (physical, virtual or cloud). This is done in the `builders` section of the configuration file.

The parameters you need to enter will depend on the type of image you want to generate. For a complete list of the mandatory and optional fields, see the builders list. Note that you can define several types of images in the same template.

When you run the hammr command to generate the images, all image formats defined in the builders section will be built at the same time.

Once the template is updated, build the images by running the command `template build`. The file specified in `--file` can either be a JSON or YAML file.

Note: For some formats, the machine image will be compressed by default. For a complete list, refer to *Machine Images Compressed by Default*.

```
$ hammr template build --file <path/filename>.yml
Validating the template file [nginx-template.yml] ...
OK: Syntax of template file [nginx-template.yml] is ok
Generating 'ami' image (1/1)
|>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| 100%: Done, created on ... |<<<<<<<<<<<<<<<<<<<<<<<<
↪<<<<<<<<<<|
OK: Generation 'ami' ok
Image URI: users/root/appliances/21/images/47
Image Id : 47
```

Note: This may take some time. A progress report is shown.

Machine Images Compressed by Default

The following tables list if the machine image will be compressed by default or not when generating your machine image with hammr.

Cloud Format	Compressed	Not Compressed
Nimbula	X	
Openstack		X
Suse Cloud		X
Eucalyptus		X
Flexiant	X	
CloudStack	X	
Abiquo		X
Azure		X
AWS		X
Outscale		X
Fujitsu K5		X

Virtual Format	Compressed	Not Compressed
OVF	X	
KVM	X	
VCenter		X
VBox	X	
RAW	X	
HyperV	X	
QCOW2	X	
VHD	X	
XEN	X	
Vagrant	X	
XenServer	X	

Container	Compressed	Not Compressed
Docker	X	
LXC	X	

Listing the Images Generated

You can check that the machine images have been created by running `image list`:

```
$ hammr image list
Getting all images and publications for [root] ...
Images:
+------+---------------+---------+------+-----------+---------------------+------+----
↪--------+------------------+
|  Id  |     Name      | Version | Rev. |  Format   |       Created       | Size |␣
↪Compressed | Generation Status |
+======+===============+=========+======+===========+=====================+======+============+======
| 1042 | generation    | 1.0     | 1    | kvm       | 2014-05-21 09:29:36 | 0B   | X ␣
↪        | Done              |
```

```
+------+--------------+---------+------+-----------+--------------------+------+----
↪--------+-------------------+
| 981  | wordpress    | 1.0     | 1    | vbox      | 2014-05-19 17:08:06 | 0B   | X ␣
↪        | Canceled          |
+------+--------------+---------+------+-----------+--------------------+------+----
↪--------+-------------------+
| 960  | nginx-muppets | 1.0     | 1    | vbox      | 2014-05-15 13:33:43 | 0B   | X ␣
↪        | Done              |
+------+--------------+---------+------+-----------+--------------------+------+----
↪--------+-------------------+

Found 3 images
No publication available
```

The table lists the image ID number, which you will need to publish the image, the name, version, revision (this is automatically increased everytime you modify the template and run `template build`), the format of the image, when it was created (date and time of the creation), if the image is compressed (not possible for all formats) and the status.

Publishing a Machine Image

In order to publish a machine image of the template you created, you must make sure that the `builders` section of the template has the necessary info for each machine image you want to publish. This includes defining the machine image you want to build as well the information for the cloud platform you want to publish to.

You will also need to set the information for your cloud account. We recommend that this information not be included in the template file, but rather set as a value that hammr will access in a seperate read-only file. For more information on creating a credential file with your cloud account information refer to the details in *Setting Your Cloud Accounts*.

The following is a YAML example of the builders section illustrating the publication to OpenStack. Note that you can incorporate details for several cloud platforms in the same configuration file. For details of the required parameters for each of the image types, refer to the documentation. You can use either YAML or JSON to create your template.

```yaml
---
builders:
- type: openstack
  hardwareSettings:
    memory: 1024
  installation:
    diskSize: 2000
  account: Openstack OW2
  tenant: opencloudware
  imageName: openstack-test
  publicImage: 'no'
  endpoint: http://ow2-04.xsalto.net:9292/v1
  keystoneEndpoint: http://ow2-04.xsalto.net:5000/v2.0
  username: test
  password: password
```

If you are using JSON:

```json
{
  "builders": [
    {
      "type": "openstack",
      "hardwareSettings": {
```

```
        "memory": 1024
      },
      "installation": {
        "diskSize": 2000
      },
      "account": "Openstack OW2",
      "tenant": "opencloudware",
      "imageName": "openstack-test",
      "publicImage": "no",
      "endpoint": "http://ow2-04.xsalto.net:9292/v1",
      "keystoneEndpoint": "http://ow2-04.xsalto.net:5000/v2.0",
      "username": "test",
      "password": "password"
    }
  ]
}
```

Publish the image(s) by running the command `image publish`:

```
$ hammr image publish --id <the id> --file <your-template>.yml
Validating the template file [/tmp/centOS.yml] ...
OK: Syntax of template file [/tmp/centOS.yml] is ok
Publishing 'ami' image (1/1)
|>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| 100%: Done, published o... |<<<<<<<<<<<<<<<<<<<<<<
↪<<<<<<<<<<<<<|
OK: Publication to 'amazon' is ok
Cloud ID : ami-25aa6752
```

Note: This may take some time. A progress report is shown.

To get the id of the machine image generated, use the command `image list`:

```
$ hammr image list
Getting all images and publications for [root] ...
Images:
+------+--------------+---------+------+-----------+---------------------+------+----
↪--------+------------------+
|  Id  |     Name     | Version | Rev. |  Format   |       Created       | Size |␣
↪Compressed | Generation Status |
+======+==============+=========+======+===========+=====================+======+============+======
| 1042 | generation   | 1.0     | 1    | kvm       | 2014-05-21 09:29:36 | 0B   | X ␣
↪        | Done              |
+------+--------------+---------+------+-----------+---------------------+------+----
↪--------+------------------+
| 1049 | generation   | 1.0     | 1    | ovf       | 2014-05-21 12:17:21 | 0B   | X ␣
↪        | In progress (2%)  |
+------+--------------+---------+------+-----------+---------------------+------+----
↪--------+------------------+
| 981  | wordpress    | 1.0     | 1    | vbox      | 2014-05-19 17:08:06 | 0B   | X ␣
↪        | Canceled          |
+------+--------------+---------+------+-----------+---------------------+------+----
↪--------+------------------+
| 960  | nginx-muppets | 1.0     | 1    | vbox      | 2014-05-15 13:33:43 | 0B   | X ␣
↪        | Done              |
+------+--------------+---------+------+-----------+---------------------+------+----
↪--------+------------------+
```

```
Found 4 images
No publication available
```

Downloading a Machine Image

You can only download images that have been compressed. You will need the image `Id` number in order to download it as follows. You must also set the location where the image should be saved. To download a machine image, use the command `image download`.

```
$ hammr image download -c ~/.hammr/credentials-UFOL.yml --id 17517 --file /tmp/test.
↪tar,gz
INFO: no username nor password provided on command line, trying credentials file
INFO: Using credentials file: /home/joris/.hammr/credentials-UFOL.yml
INFO: Using url https://factory.usharesoft.com/api
Searching image with id [17517] ...
Status: 100% |>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
↪>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| Time: 0:01:11
OK: Image downloaded
```

CHAPTER 9

Importing and Exporting

Hammr has a notion of importing and exporting templates. After creating a template from the JSON or YAML configuration file; this template can be exported as an archive. The archive will include the template file as well as any bundled software that was initially uploaded as part of the template creation.

The archive can then be used to import the template into another UForge Server instance.

The `stack` section of a template can include bundles of software and configuration information. This information may be stored locally on a filesystem or available via URLs. In some cases, when another user creates a template from the same template file (JSON or YAML), the custom software and/or configuration files may not be reachable or present. By creating an archive (using export) ensure that all relevant software for creating the template is available.

Exporting a Template

To illustrate exporting a template, let's start from scratch. We will create a template, get the ID and export it with hammr.

So first lets create a new template with the YAML file `centoscore-template.yml`.

Note: You can also use JSON.

```
---
stack:
  name: CentOS Core
  version: '6.4'
  os:
    name: CentOS
    version: '6.4'
    arch: x86_64
    profile: Minimal
  config:
  - name: firstboot1.sh
```

```
    source: http://myconfig.site.com/config/firstboot1.sh
    type: bootscript
    frequency: firstboot
  - name: firstboot0.sh
    source: http://myconfig.site.com/config/firstboot1.sh
    type: bootscript
    frequency: firstboot
```

If you are using JSON:

```
{
  "stack" : {
    "name" : "CentOS Core",
    "version" : "6.4",
    "os" : {
      "name" : "CentOS",
      "version" : "6.4",
      "arch" : "x86_64"
      "profile" : "Minimal"
    },
    "config" : [ {
      "name" : "firstboot1.sh",
      "source" : "http://myconfig.site.com/config/firstboot1.sh",
      "type" : "bootscript",
      "frequency" : "firstboot"
    }, {
      "name" : "firstboot0.sh",
      "source" : "http://myconfig.site.com/config/firstboot1.sh",
      "type" : "bootscript",
      "frequency" : "firstboot"
    } ]
  }
}
```

```
$ hammr template create --file centoscore-template.yml
```

Now that the template is created we need to get the `Id` of the template you want to export. To do so, list the templates with the command `template list`:

```
$ hammr template list
+-----+------------------------------+---------+---------------------+----------------
↪-----+---------------------+--------+---------+-----+--------+
| Id  |             Name             | Version |         OS          |       Created ␣
↪     |    Last modified    | # Imgs | Updates | Imp | Shared |
+=====+==============================+=========+=====================+=================
| 669 | CentOS Core                  | 1.0     | CentOS 6.4 x86_64   | 2014-04-25␣
↪13:55:19 | 2014-05-09 13:24:59 | 0      | 0       |     |        |
+-----+------------------------------+---------+---------------------+----------------
↪-----+---------------------+--------+---------+-----+--------+
```

In this case the `Id` is `669`. To export the template, run the command `template export`:

```
$ hammr template export --id 669 --file /tmp/centos-core-archive.tar.gz
Exporting template with id [669] :
100%|##############################################################################
↪#|
Downloading archive...
```

```
OK: Download complete of file [/tmp/centos-core-archive.tar.gz]
```

Now if you uncompress the archive, you will find a file `template.yml`, which is the template YAML configuration file and a sub-directory `config` containing the two boot scripts.

Note: If the command `template export` has `--outputFormat json` argument, the file `template.yml` in the result archive will be replaced by file `template.json`.

If you open the `template.yml` file, then you will notice that there is additional information added, including:

- `pkgs`: this contains all the packages that are added by the os profile `Minimal`
- `updateTo`: this is the date that the template initially created. This ensures that if you re-import this template (the creation date might be different) and build a machine image, the machine image will be identical to any machine image built from the original template
- `installation`: adds the default installation parameters.

Importing a Template

You can import a template based on a tar.gz archive file by using the command `template import`. This will import the archive, which contains the JSON or YAML file and binaries of the template.

```
$ hammr template import --file /tmp/centos-core-archive.tar.gz
Importing template from [/tmp/centos-core-archive.tar.gz] archive ...
100%|##################################################################################
↪#|
OK: Template import: DONE
Template URI: users/root/appliances/22
Template Id : 22
```

CHAPTER 10

Templates Specification

Templates contain all the information used to create stacks; build machine images and publish them to the target platform. They are JSON or YAML files, passed as a parameter to the hammr command-line.

A template has two main parts:

- `stack`: defines the packages, files and configuration scripts of the machine image to build.
- `builders`: an array defining the format of the machine images to build.

Stack

Within a template, the stack section describes the packages, files and configuration information required to be added when building a machine image. It can also contain low level installation information (for example keyboard settings, partitioning, timezone etc) to be configured as part of the build or prompted during the first boot of an instance using the machine image.

The definition of a `stack` section when using YAML is:

```
---
stack:
  # the stack definition goes here
```

If you are using JSON:

```
{
        "stack": {
            ...the stack definition goes here.
        }
}
```

The valid keys to use within a stack are:

- `name` (mandatory): a string providing the name of the stack
- `version` (mandatory): a string providing the version of the stack

- `description` (optional): a string providing a description of what the stack oes
- `os` (mandatory): an object providing the operating system to use when building the machine image. You must have access to this operating system in UForge. This object may include specific packages to install from the operating system repository. For more information, refer to the *os* sub-section.
- `bundles` (optional): an array of objects describing any software bundles (can be native packages, tarballs, jars, wars etc) to upload and use when building the machine image. For more information, refer to the *bundles* sub-section.
- `installation` (optional): an object providing low-level installation or first boot options. Some options can be pre-configured as part of the build or prompted by the end-user to provide when provisioning an instance from the machine image. For more information, refer to the *installation* sub-section.
- `config` (optional): an array of objects describing any configuration scripts to execute when an instance is booted from the machine image. For more information, refer to the *config* sub-section.

Stack sub-sections are:

os

Within a `stack`, the `os` sub-section describes the operating system to use when building the machine image. This includes the operating system version, architecture and the os profile to use. The os profile is a pre-determined group of packages that will be installed as part of the machine image build. Extra packages can be specified to include in the build that are available in the operating system repository, and the build date can be set to get the latest updates, or roll-back. For more information on os package updates, refer to *Package Updates*.

To use a particular operating system, you must have access to it in the UForge server you are using. To determine which operating systems are available, use the `os list` command (please refer *Command-Line* for more information).

The definition of an `os` section when using YAML is:

```
---
os:
  # the os definition goes here
```

If you are using JSON:

```
"os": {
    ...the os definition goes here.
}
```

The valid keys to use within the os object are:

- `arch` (mandatory): a string providing the architecture to use
- `name` (mandatory): a string providing the name of the operating system to use
- `pkgs` (optional): an array providing any extra packages to install (see pkgs key sub-section for more information)
- `profile` (mandatory): a string providing which operating system profile to use
- `updateTo` (optional): a string providing the date and time where package versions should be calculated (determines which package versions to use to * calculate the package dependency tree)
- `version` (mandatory): a string providing the version of the operating system to use

Sub-Sections

The `os` sub-sections are:

pkgs

Within the *os* section, the `pkgs` sub-section is an array of objects describing any extra packages that should be installed as part of the machine image build. Any package information provided in this section must exist in the corresponding operating system repository, otherwise this will result in a build failure.

The definition of a `pkgs` section when using YAML is:

```
---
pkgs:
- # the list of packages goes here.
```

If you are using JSON:

```
"pkgs": [
    ...the list of packages goes here.
]
```

The valid keys to use within the pkgs object are:

- `arch` (optional): a string providing architecture to use
- `fullName` (optional): a string providing the name, version, release and architecture information. If used, the mandatory name key is not required.
- `name` (mandatory): a string providing the name of the package to use
- `release` (optional): a string providing the release of the package to use
- `version` (optional): a string providing the version of the package to use

When `name` is used on its own, the version, release and arch is determined when the machine image is being built. This information is determined during the package dependency phase of the build. The package dependency phase uses the created date of the stack within the UForge server to calculate the correct versions of packages. This date can be overridden by the `updateTo` key in the os section. Any missing packages required by the stack are also added to ensure any dependencies are met.

In the case where `version`, `release` and `arch` (or `fullName`) is used, then the version determined by the created stack date (or updateTo date) is overridden by the version details provided. This is known as making the package sticky. Note, that any updates available for this package will NOT be used in this case.

Examples

Basic Example

The following example uses CentOS 6.4 64 bit operating system for the template and adding the packages `php`, `php-cli`, `php-common` and `php-mysql`. Note that only the name is provided. The final version and release of these packages is determined during the build of the machine image.

If you are using YAML:

```
---
os:
  name: CentOS
  version: '6.4'
  arch: x86_64
  profile: Minimal
  pkgs:
  - name: php
  - name: php-cli
  - name: php-common
  - name: php-mysql
```

If you are using JSON:

```
{
  "os": {
    "name": "CentOS",
    "version": "6.4",
    "arch": "x86_64",
    "profile": "Minimal",
    "pkgs": [
      {
        "name": "php"
      },
      {
        "name": "php-cli"
      },
      {
        "name": "php-common"
      },
      {
        "name": "php-mysql"
      }
    ]
  }
}
```

Adding a Version and Release

By adding `version`, `release` and `arch` or `fullName`, during the build this specific version is used regardless of any build date (`updateTo`) set in the `os` section. This is called making the package "sticky".

If you are using YAML:

```
---
os:
  name: CentOS
  version: '6.4'
  arch: x86_64
  profile: Minimal
  pkgs:
  - name: php
    version: 5.5.3
    release: 23.el6_4
    arch: i686
  - name: php-cli
```

```
    version: 5.5.3
    release: 23.el6_4
    arch: i686
  - fullName: php-common-5.5.3-23.el6_4-i686.rpm
  - fullName: php-mysql-5.5.3-23.el6_4-i686.rpm
```

If you are using JSON:

```
{
  "os": {
    "name": "CentOS",
    "version": "6.4",
    "arch": "x86_64",
    "profile": "Minimal",
    "pkgs": [
      {
        "name": "php",
        "version": "5.5.3",
        "release": "23.el6_4",
        "arch": "i686"
      },
      {
        "name": "php-cli",
        "version": "5.5.3",
        "release": "23.el6_4",
        "arch": "i686"
      },
      {
        "fullName": "php-common-5.5.3-23.el6_4-i686.rpm"
      },
      {
        "fullName": "php-mysql-5.5.3-23.el6_4-i686.rpm"
      }
    ]
  }
}
```

Examples

Basic Example

The following example describes using CentOS 6.4 64 bit operating system for the template. The profile `Minimal` is used, which automatically adds a pre-determined group of packages to the template.

If you are using YAML:

```
---
os:
  name: CentOS
  version: '6.4'
  arch: x86_64
  profile: Minimal
```

If you are using JSON:

```
{
  "os": {
    "name": "CentOS",
    "version": "6.4",
    "arch": "x86_64",
    "profile": "Minimal"
  }
}
```

The name, version, arch and profile values for an operating system can be found by using the command `os list`. This lists all the operating systems you have access to.

Specifying a Build Date/Time

By using the `updateTo` key will specify the date and time to calculate the version and release of all the packages to use in the template during the build phase of a machine image. This allows you to roll-back or update the operating system packages used. If no date is provided, then the date the template is created is used. The example below sets the date to 14 May 2014 00:00 UTC. Note that timezone must respect General Time Zone format.

If you are using YAML:

```
---
os:
  name: CentOS
  version: '6.4'
  arch: x86_64
  profile: Minimal
  updateTo: 2014-05-14 00:00 UTC
```

If you are using JSON:

```
{
  "os": {
    "name": "CentOS",
    "version": "6.4",
    "arch": "x86_64",
    "profile": "Minimal"
    "updateTo": "2014-05-14 00:00 UTC"
  }
}
```

You can still use a date without time information, in such case time will be interpreted by server as midnight in server's own timezone (UTC by default).

If you are using YAML:

```
---
os:
  name: CentOS
  version: '6.4'
  arch: x86_64
  profile: Minimal
  updateTo: '2014-05-14'
```

If you are using JSON:

```
{
  "os": {
    "name": "CentOS",
    "version": "6.4",
    "arch": "x86_64",
    "profile": "Minimal"
    "updateTo": "2014-05-14"
  }
}
```

bundles

Within a `stack`, the `bundles` sub-section describes any custom software to be added to the filesystem of the machine image during the build phase. Software bundles can contain any file, archive or native package. Native packages can be installed and archives can be uncompressed as part of this process.

The definition of a `bundles` section when using YAML is:

```
---
bundles:
- # the list of bundles goes here.
```

If you are using JSON:

```
"bundles": {
    ...the list of bundles goes here.
}
```

The valid keys to use within a bundle are:

- `description` (optional): a string describing what the bundle does or contains.
- `destination` (optional): a string providing the target directory where to add the files in the filesystem.
- `files` (mandatory): an array of objects describing the files, archives or packages contained in the bundle (each element can have another files key). See the *files* sub-section for available keys.
- `license` (optional): an object providing the license information for the bundle. See the *license* sub-section for available keys.
- `name` (mandatory): a string providing the name of the bundle.
- `version` (mandatory): a string providing the version of the bundle.
- `shortTag` (optional): a string providing the short tag for the bundle.
- `category` (optional): a string providing the category name for the bundle.
- `maintainer` (optional): a string providing the maintainer for the bundle (if not provided, the user loginName will be used).
- `website` (optional): a string providing the website URL for the bundle or maintainer.
- `restrictionRule` (optional): a string which indicates the "restriction rule" for a bundle. A restriction rule lists which distributions and/or target formats the bundle is designed for.
- `sourceLogo` (optional): a string providing the location of where to get the file. This can be a filesystem path (absolute or relative) or an URL.

The destination string that describes where to add the files in the bundle is ignored for native packages that have the option to be installed during the build process.

Sub-sections

Bundle sub-sections are:

files

Within a `bundle`, the `files` sub-section describes the list of files, binaries, archives or native packages that are part of the bundle. Within the `files` you can also list a folder.

Note: If you list a folder in `files` sub-section, it can have its own `files` sub-section to provide more information for a specific file in the folder (i.e. rights, owner:group, symlink etc)

The definition of a `files` section when using YAML is:

```
---
files:
- # the list of files goes here.
```

If you are using JSON:

```
{
        "files": [
            ...the list of files goes here.
        ]
}
```

The valid keys to use within a file are:

- `destination` (optional): a string providing the destination path where to install the file on the machine image filesystem. This overrides any destination install path provided in the bundle section.
- `ownerGroup` (optional): a string corresponding to the owner:group to set during generation for the file (tag key must be equals to softwarefile. If not provided, "root:root" will be used by default).
- `rights` (optional): a string corresponding to the rights (i.e. 755) to set during generation for the file (tag key must be equals to softwarefile. If not provided, "755" will be used by default).
- `symlink` (optional): a string providing the location of the symlink to create during generation for the file (tag key must be equals to softwarefile).
- `bootOrder` (optional): an integer providing the boot order. (tag key must be equals to bootscript).
- `bootType` (optional): a string providing the script type, firstboot or everyboot (tag key must be equals to bootscript).
- `tag` (optional): a string describing the type of the file provided (softwarefile, ospkg or bootscript). ospkg and bootscript can only be in the first level files section (if not provided, the file will be considered as a softwarefile)
- `extract` (optional): a boolean describing whether to uncompress/extract the archive file during the build process. This flag can only be used if the file is an archive, otherwise this flag is ignored.
- `install` (optional): a boolean describing whether to install the native package as part of the build. It can only be used for native packages. If false, then the native package will be added to the filesystem as a file described by the destination. Otherwise the package will be treated like any other native package – package dependencies will be verified and installed. If the file is not a native package then this flag is ignored.
- `md5sum` (optional): a string providing a md5sum checksum.

- `name` (mandatory): a string providing the name of the bundle.
- `params` (optional): a string providing any parameters to execute with the binary file as part of the generation process. These parameters are ignored if the file is not a binary (.msi, .exe etc)
- `source` (mandatory): a string providing the location of where to get the file. This can be a filesystem path (absolute or relative) or an URL.
- `files` (optional): an array of objects describing the files, archives or packages contained in the file representing a folder (file must represents a folder and tag key must be equals to softwarefile and all files in it can only be tagged as softwarefile).

Examples

Basic Example

The following example shows how to declare a set of files to uploaded as part of a bundle.

If you are using YAML:

```
---
files:
- name: wordpress.zip
  source: http://wordpress.org/wordpress-3.5.zip
```

If you are using JSON:

```
{
  "files": [
    {
      "name": "wordpress.zip",
      "source": "http://wordpress.org/wordpress-3.5.zip"
    }
  ]
}
```

Example of a Folder in Files Sub-section

The following example shows how to declare a folder to be uploaded as part of a bundle. All the files within the declared bundle will be uploaded.

Note: You cannot upload the same source folder with two different names. In the end, the source folder and files will only be uploaded once.

If you are using YAML:

```
---
files:
- name: folder
  source: "/usr/local/folder"
```

If you are using JSON:

```
{
  "files": [
    {
      "name": "folder",
      "source": "/usr/local/folder"
    }
  ]
}
```

Example of a Folder in Files Sub-section with its own Files Sub-section

The following example shows how to declare a folder to be uploaded as part of a bundle. All the files within the declared bundle will be uploaded. A folder can have a Files Sub-section to to add more information for a file in it like rights, owner:group etc.

Note: You cannot upload the same source folder with two different names. In the end, the source folder and files will only be uploaded once.

If you are using YAML:

```
---
  files:
  - name: "folder"
        tag: "softwarefile"
        source: "/usr/local/folder"
        files:
        - name: "filename1.zip"
          ownerGroup: "root:root"
          rights: "755"
          symlink: "/tmp/filename1_symlink.zip"
          tag: "softwarefile"
          source: "/usr/local/folder/filename1.zip"
          files: []
        - name: "folder2"
          ownerGroup: "root:root"
          rights: "765"
          tag: "softwarefile"
          source: "/usr/local/folder/folder2"
          files:
          - name: "folder3"
            ownerGroup: "root:root"
            rights: "755"
            symlink: "/tmp/folder3_symlink"
            tag: "softwarefile"
            source: "/usr/local/folder/folder2/folder3"
            files: [
                - name: "filename3.zip"
                  ownerGroup: "root:root"
                  rights: "765"
                  tag: "softwarefile"
                  source: "/usr/local/folder/folder2/folder3/filename3.zip"
                  files: []
```

If you are using JSON:

```
{
  "files": [
    {
      "name": "folder",
      "tag" : "softwarefile",
      "source": "/usr/local/folder",
      "files": [
                    {
                          "name": "filename1.zip",
                          "ownerGroup" : "root:root",
                          "rights" : "755",
                          "symlink" : "/tmp/filename1_symlink.zip",
                          "tag" : "softwarefile",
                          "source": "/usr/local/folder/filename1.zip",
                          "files" : [ ]
                    },
                    {
                          "name": "folder2",
                          "ownerGroup" : "root:root",
                          "rights" : "765",
                          "tag" : "softwarefile",
                          "source": "/usr/local/folder/folder2",
                          "files": [
                            {
                                  "name": "folder3",
                                  "ownerGroup" : "root:root",
                                  "rights" : "755",
                                  "symlink" : "/tmp/folder3_symlink",
                                  "tag" : "softwarefile",
                                  "source": "/usr/local/folder/folder2/folder3",
                                  "files": [
                                    {
                                          "name": "filename3.zip",
                                          "ownerGroup" : "root:root",
                                          "rights" : "765",
                                          "tag" : "softwarefile",
                                          "source": "/usr/local/folder/folder2/
↪folder3/filename3.zip",
                                          "files" : [ ]
                                    }
                                  ]
                            }
                          ]
                    }
                  ]
    }
  ]
}
```

Overriding Bundle Destination

The bundle via destination provides the global install path for all the files. This example shows how you can add a file to another directory in the filesystem, effectively overriding the default destination directory.

If you are using YAML:

```
---
files:
- name: wordpress.zip
  source: http://wordpress.org/wordpress-3.5.zip
  destination: "/usr/local/wordpress"
```

If you are using JSON:

```
{
  "files": [
    {
      "name": "wordpress.zip",
      "source": "http://wordpress.org/wordpress-3.5.zip",
      "destination": "/usr/local/wordpress"
    }
  ]
}
```

Extracting Archives

The example uses the extract key to automatically extract the archive file:

If you are using YAML:

```
---
files:
- name: wordpress.zip
  source: http://wordpress.org/wordpress-3.5.zip
  destination: "/usr/local/wordpress"
  extract: true
```

If you are using JSON:

```
{
  "files": [
    {
      "name": "wordpress.zip",
      "source": "http://wordpress.org/wordpress-3.5.zip",
      "destination": "/usr/local/wordpress",
      "extract": true
    }
  ]
}
```

Installing or Placing Native Packages

The example declares a native package to be added to the bundle. The install key is used to tell the build process not to install the package, but to add it to the filesystem in the destination directory.

If you are using YAML:

```
---
files:
- name: "mypackage.rpm"
  source: "/home/joris/demo/mypackage-3.1.rpm"
```

```
  destination: "/usr/local/rpms"
  tag: "softwarefile"
  install: false
```

If you are using JSON:

```
{
  "files": [{
      "name": "mypackage.rpm",
      "source": "/home/joris/demo/mypackage-3.1.rpm",
      "destination": "/usr/local/rpms",
          "tag" : "softwarefile",
      "install": false
    }
  ]
}
```

If install is set to `true`, then the package is installed as a native package (including package dependency checking) and then destination information is ignored.

Note: A native package is different than a repository package. See next example for the differences

Introduced simple file, repository package and bootscript

The example declares a repository package, a bootscript and a file to be added to the bundle. The tag key is used to tell what kind of file the section represents (if tag is not specified, it will be considered as a simple file to be uploaded).

If you are using YAML:

```
---
files:
- name: "filename1.zip"
  ownerGroup: "root:root"
  rights: "755"
  symlink: "/tmp/filename1_symlink.zip"
  tag: "softwarefile"
  source: "/usr/local/folder/filename1.zip"
  md5sum: "1367cd10d6ce432cc44b4dc4bb2c4b01"
  sha256sum: "72bc4caf81f5d9943ba32bf9703262e71ac256ca0308dfdbcc3c0b78a5d01cd4"
  files: []
- name: "iotop-0.6-2.el7.noarch.rpm"
  tag: "ospkg"
  source: "/usr/local/folder/iotop-0.6-2.el7.noarch.rpm"
  files: []
- name: "bootscriptMysoftware.sh"
  bootOrder: 1
  bootType: "firstboot"
  tag: "bootscript"
  source: "/usr/local/folder/bootscriptMysoftware.sh"
  install: true
  md5sum: "a510c417e546c67a61c720a3696ef87c"
  sha256sum: "a7ae23c18a84338e9425a68a72c1b7cf66ea6ed30bd142ee0a824d6bf02e67e1"
  files: []
```

If you are using JSON:

```
{
  "files": [{
          "name": "filename1.zip",
          "ownerGroup" : "root:root",
          "rights" : "755",
          "symlink" : "/tmp/filename1_symlink.zip",
          "tag" : "softwarefile",
          "source": "/usr/local/folder/filename1.zip",
          "files" : [ ]
    }, {
          "name" : "iotop-0.6-2.el7.noarch.rpm",
          "tag" : "ospkg",
          "source" : "/usr/local/folder/iotop-0.6-2.el7.noarch.rpm",
          "files" : [ ]
    }, {
          "name" : "bootscriptMysoftware.sh",
          "bootOrder" : 1,
          "bootType" : "firstboot",
          "tag" : "bootscript",
          "source" : "/usr/local/folder/bootscriptMysoftware.sh",
          "install" : true,
          "md5sum" : "a510c417e546c67a61c720a3696ef87c",
          "sha256sum" :
↪"a7ae23c18a84338e9425a68a72c1b7cf66ea6ed30bd142ee0a824d6bf02e67e1",
          "files" : [ ]
    }
  ]
}
```

The `source` section for a file tagged as `ospkg` must be here but can be a false source because it doesn't matter. The repository package will be searched by the distribution and the fullname To get the fullname of a repository package, See the *Searching for Packages*. During export or create, a file tagged as `ospkg` will be search by fullname in the unique distribution given in `oses` section of the bundle.

Using Parameters for Binaries

The example declares a binary file to be added to the bundle. The params key is used to provide a set of parameters that are used to execute the binary.

If you are using YAML:

```
---
files:
- name: mybinary.exe
  source: "/home/joris/demo/mybinary.exe"
  params: "--silent"
```

If you are using JSON:

```
{
  "files": [
    {
      "name": "mybinary.exe",
      "source": "/home/joris/demo/mybinary.exe",
      "params": "--silent"
    }
```

```
  ]
}
```

Warning: Hammr only supports windows binaries to be executed with parameters (.exe and .msi). For linux, use the *config* section to declare boot scripts.

license

Within a `bundle`, the `license` sub-section describes the license information or EULA for the files that make up the bundle.

The definition of a `license` section when using YAML is:

```
---
license:
  # the license declaration goes here.
```

If you are using JSON:

```
"license": {
    ...the license declaration goes here.
}
```

The valid keys to use within a license are:

- `name` (mandatory): a string providing the name of the license.
- `source` (mandatory): a string providing the location of where to get the license. This can be a filesystem path or URL.

Example

The following example shows how to declare a license for a bundle.

If you are using YAML:

```
---
license:
  name: license.html
  source: "/home/joris/demo/apache-license.html"
```

If you are using JSON:

```
{
  "license": {
    "name": "license.html",
    "source": "/home/joris/demo/apache-license.html"
  }
}
```

Examples

Basic Example

The following example describes the mandatory information in a bundle to be uploaded and used in the template. All the files described in the bundle are placed in the `/tmp/wordpress` directory.

If you are using YAML:

```
---
bundles:
- name: "wordpress"
  version: "3.5"
  destination: "/tmp/wordpress"
  maintainer: "wordpress"
  files:
  - # add files definition here  (see :ref:`stack-bundle-files` sub-section)
- name: "wordpress language pack"
  version: "3.5"
  destination: "/tmp/wordpress"
  maintainer: "wordpress"
  files:
  - # add files definition here  (see :ref:`stack-bundle-files` sub-section)
```

If you are using JSON:

```
{
  "bundles": [
    {
      "name": "wordpress",
      "version": "3.5",
      "destination": "/tmp/wordpress",
      "maintainer": "wordpress",
      "files": [
          ...add files definition here (see :ref:`stack-bundle-files` sub-section)
      ]
    },
    {
      "name": "wordpress language pack",
      "version": "3.5",
      "destination": "/tmp/wordpress",
      "maintainer": "wordpress",
      "files": [
          ...add files definition here (see :ref:`stack-bundle-files` sub-section)
      ]
    }
  ]
}
```

Adding a Description and License

The following example show how you can add license information and a description to the bundle.

If you are using YAML:

```
---
bundles:
- name: "wordpress"
  version: "3.5"
```

```
  description: "The wordpress files from wordpress.org"
  destination: "/tmp/wordpress"
  maintainer: "wordpress"
  files:
  - # add files definition here (see :ref:`stack-bundle-files` sub-section)
  license:
  - # add license definition here (see :ref:`stack-bundle-license` sub-section)
```

If you are using JSON:

```
{
  "bundles": [
    {
      "name": "wordpress",
      "version": "3.5",
      "description": "The wordpress files from wordpress.org",
      "destination": "/tmp/wordpress",
      "maintainer": "wordpress",
      "files": [
          ...add files definition here (see :ref:`stack-bundle-files` sub-section)
      ],
      "license": {
          ...add license definition here (see :ref:`stack-bundle-license` sub-section)
      }
    }
  ]
}
```

Adding a Website, Category and sourceLogo

The following example show how you can add website, category and logo information to the bundle.

If you are using YAML:

```
---
bundles:
- name: "wordpress"
  version: "3.5"
  description: "The wordpress files from wordpress.org"
  destination: "/tmp/wordpress"
  maintainer: "wordpress"
  website: "https://fr.wordpress.org/"
  category: "Blogging"
  files:
  - # add files definition here (see :ref:`stack-bundle-files` sub-section)
  license:
  - # add license definition here (see :ref:`stack-bundle-license` sub-section)
  sourceLogo: "/tmp/wordpress.png"
```

If you are using JSON:

```
{
  "bundles": [
    {
      "name": "wordpress",
      "version": "3.5",
```

```
      "description": "The wordpress files from wordpress.org",
      "destination": "/tmp/wordpress",
      "maintainer": "wordpress",
      "website": "https://fr.wordpress.org/",
      "category": "Blogging",
      "files": [
          ...add files definition here (see :ref:`stack-bundle-files` sub-section)
      ],
      "license": {
          ...add license definition here (see :ref:`stack-bundle-license` sub-section)
      },
      "sourceLogo": "/tmp/wordpress.png"
    }
  ]
}
```

To get the list of categories available, run command:

```
$ hammr bundle categories
```

Adding Restriction Rules

The following example shows how you can set rules to restriction the bundle to specific distributions and/or target formats. For a list of supported distribution and/or target formats refer to *os* and/or *os*.

The restriction rule is represented by a `logical` expression. The format of simple expression is: `Object#field=value` or `Object#field!=value`.

- object is `Distribution` or `TargetFormat`
- field is `family`, `pkgType`, `name`, `version` or `arch` for Distribution object (see *os* and os list command to find values for field)
- field is `name` or `type` for TargetFormat object (refer to *Builders* section to get values for `name` field. Values for `type` field are cloud, virtual, physical or container)
- value is the value you want to match with the fields (e.g. CentOS for Distribution name, linux for Distribution family, x86_64 for Distribution arch, VirtualBox for TargetFormat name, cloud for TargetFormat type...)
- logical operator is `||` for OR and `&&` for AND
- carriage return is not authorized

In the following example, the bundle is designed for (CentOS 6 x86_64 or CentOS 7 x86_64) or (all distribution if image is generated for VirtualBox)

If you are using YAML:

```
---
bundles:
- name: "wordpress"
  version: "3.5"
  description: "The wordpress files from wordpress.org"
  destination: "/tmp/wordpress"
  maintainer: "wordpress"
  website: "https://fr.wordpress.org/"
  category: "Blogging"
  restrictionRule: "(Distribution#name=CentOS && Distribution#arch=x86_64 &&␣
↪(Distribution#version=7 || Distribution#version=6)) || TargetFormat#name=VirtualBox"
```

```
  files:
  - # add files definition here (see :ref:`stack-bundle-files` sub-section)
  license:
  - # add license definition here (see :ref:`stack-bundle-license` sub-section)
  sourceLogo: "/tmp/wordpress.png"
```

If you are using JSON:

```
{
  "bundles": [
    {
      "name": "wordpress",
      "version": "3.5",
      "description": "The wordpress files from wordpress.org",
      "destination": "/tmp/wordpress",
      "maintainer": "wordpress",
      "website": "https://fr.wordpress.org/",
      "category": "Blogging",
      "restrictionRule" : "(Distribution#name=CentOS && Distribution#arch=x86_64 &&␣
↪(Distribution#version=7 || Distribution#version=6)) || TargetFormat#name=VirtualBox
↪",
      "files": [
          ...add files definition here (see :ref:`stack-bundle-files` sub-section)
      ],
      "license": {
          ...add license definition here (see :ref:`stack-bundle-license` sub-section)
      },
      "sourceLogo": "/tmp/wordpress.png"
    }
  ]
}
```

Adding Third Party Software

Within a template, you can include your own software and licenses, including bootscripts. The following is an example stack which uploads x file to a template

If you are using YAML:

```
---
bundles:
- name: "wordpress"
  version: "3.5"
  description: "The wordpress files from wordpress.org"
  destination: "/tmp/wordpress"
  maintainer: "wordpress"
  website: "https://fr.wordpress.org/"
  category: "Blogging"
  restrictionRule: "Distribution#name=CentOS && Distribution#arch=x86_64 &&␣
↪Distribution#version=7"
  files:
  - name: "filename1.zip"
    ownerGroup: "root:root"
    rights: "755"
    symlink: "/tmp/filename1_symlink.zip"
    tag: "softwarefile"
```

```
    source: "/usr/local/folder/filename1.zip"
    md5sum: "1367cd10d6ce432cc44b4dc4bb2c4b01"
    sha256sum: "72bc4caf81f5d9943ba32bf9703262e71ac256ca0308dfdbcc3c0b78a5d01cd4"
    files: []
  - name: "filename2.txt"
    ownerGroup: "root:root"
    rights: "755"
    tag: "softwarefile"
    source: "/usr/local/folder/filename2.txt"
    files: []
  - name: "iotop-0.6-2.el7.noarch.rpm"
    tag: "ospkg"
    source: "/usr/local/folder/iotop-0.6-2.el7.noarch.rpm"
    files: []
  - name: "bootscriptMysoftware.sh"
    bootOrder: 1
    bootType: "firstboot"
    tag: "bootscript"
    source: "/usr/local/folder/bootscriptMysoftware.sh"
    install: true
    md5sum: "a510c417e546c67a61c720a3696ef87c"
    sha256sum: "a7ae23c18a84338e9425a68a72c1b7cf66ea6ed30bd142ee0a824d6bf02e67e1"
    files: []
  license:
  - # add license definition here (see :ref:`stack-bundle-license` sub-section)
  sourceLogo: "/tmp/wordpress.png"
```

If you are using JSON:

```
{
  "bundles": [
    {
      "name": "wordpress",
      "version": "3.5",
      "description": "The wordpress files from wordpress.org",
      "destination": "/tmp/wordpress",
      "maintainer": "wordpress",
      "website": "https://fr.wordpress.org/",
      "category": "Blogging",
      "restrictionRule" : "Distribution#name=CentOS && Distribution#arch=x86_64 &&␣
↪Distribution#version=7",
      "files": [{
                "name": "filename1.zip",
                "ownerGroup" : "root:root",
                "rights" : "755",
                "symlink" : "/tmp/filename1_symlink.zip",
                "tag" : "softwarefile",
                "source": "/usr/local/folder/filename1.zip",
                "files" : [ ]
          }, {
                "name": "filename2.txt",
                "ownerGroup" : "root:root",
                "rights" : "755",
                "tag" : "softwarefile",
                "source": "/usr/local/folder/filename2.txt",
                "files" : [ ]
          }, {
                "name" : "iotop-0.6-2.el7.noarch.rpm",
```

```
                "tag" : "ospkg",
                "source" : "/usr/local/folder/iotop-0.6-2.el7.noarch.rpm",
                "files" : [ ]
          }, {
                "name" : "bootscriptMysoftware.sh",
                "bootOrder" : 1,
                "bootType" : "firstboot",
                "tag" : "bootscript",
                "source" : "/usr/local/folder/bootscriptMysoftware.sh",
                "install" : true,
                "md5sum" : "a510c417e546c67a61c720a3696ef87c",
                "sha256sum" :
↪"a7ae23c18a84338e9425a68a72c1b7cf66ea6ed30bd142ee0a824d6bf02e67e1",
                "files" : [ ]
      } ],
      "license": {
          ...add license definition here (see :ref:`stack-bundle-license` sub-section)
      },
      "sourceLogo": "/tmp/wordpress.png"
    }
  ]
}
```

installation

Within a *Stack*, the `installation` sub-section describes questions that are normally related to the installation of an operating system. This includes root password, keyboard settings, timezone, and partitioning. These questions are only asked once as part of the operating system installation; consequently decided by the person when building machine images manually. Hammr provides a mechanism that allows some of the installation questions to be asked as part of the first-boot when provisioning an instance from the machine image. This makes any machine image built by hammr to be more flexible, for example if you have a team in the UK and another team in France then their keyboard settings are most likely to be QWERTY and AZERTY respectively. Allowing the end-user to choose the keyboard settings as part of first-boot can help resolve hours of frustration.

The definition of a `pkgs` section when using YAML is:

```
---
installation:
  # the installation definition goes here.
```

If you are using JSON:

```
"installation": {
    ...the installation definition goes here.
}
```

The valid keys to use within an installation are:

- `diskSize` (optional): an integer value (in MB) providing the disk size of the machine image. This value is ignored if an advanced partitioning table is provided (see *partitioning*)
- `displayLicense` (optional): a boolean value to display any EULA during the first boot of a provisioned instance (includes operating system EULA and any license information provided in the *bundles* section of the stack). If the value is `false` then no license information is displayed. If `displayLicense` is not used, then by default all license information is displayed during first boot.

- `internetSettings` (optional): a string providing the network settings. Possible values `basic`, `ask` or `configure`. If no value is provided, is set to `basic` by default. Refer to the *internetSettings* sub-section for more information.
- `kernelParams` (optional): an array of strings providing the kernel parameters to use. These parameters are used when provisioning an instance from the machine image. If no kernel parameters are provided, the `rhbg` and `quiet` parameters are set by default
- `keyboard` (optional): a string providing the keyboard layout to use. If no keyboard setting is provided, then during first boot the keyboard setting is prompted. See *keyboard* for all available values for keyboard
- `partitioning` (optional): an array of objects describing an advanced partitioning table. Refer to *partitioning* sub-section for more information.
- `rootUser` (optional): an object describing the configuration information of the root user (or primary administrator). If `rootUser` is not provided, then during first boot the root user password is prompted
- `swapSize` (optional): an integer value (in MB) providing the swap size to be allocated. This value is ignored if an advanced partitioning table is provided (see *partitioning*)
- `timezone` (optional): a string providing the timezone to use. If no timezone is provided, then during first boot the timezone is prompted. See *timezone* for all available values for timezone.
- `firewall` (optional): a boolean to enable or disable the firewall service. If no firewall is given, then the firewall is asked during installation.
- `welcomeMessage` (optional): a string providing a welcome message displayed during the first boot of a provisioned instance
- `seLinuxMode` (optional): a string indicating the SELinux mode (see *selinux*)

Sub-sections

The `installation` sub-sections are:

keyboard

The following is a list of all the accepted keyboard values that can be used for the keyboard key in an *installation* sub-section of the stack.

You do not need to enter the full name; only the short form is required. For example, for `Danish` use:

If you are using YAML:

```
---
installation:
  keyboard: dk
```

If you are using JSON:

```
{
  "installation": {
    "keyboard": "dk"
  }
}
```

Available Keyboard Values

- Arabic (azerty) `ar-azerty`
- Arabic (azerty/digits) `ar-azerty-digits`
- Arabic (digits) `ar-digits`
- Arabic (qwerty) `ar-qwerty`
- Arabic (qwerty/digits) `ar-qwerty-digits`
- Belgian (be-latin1) `be-latin1`
- Bengali (Inscript) `ben`
- Bengali (Probhat) `ben-probhat`
- Brazilian (ABNT2) `br-abnt2`
- Bulgarian (Phonetic) `bg_pho-utf8`
- Bulgarian `bg_bds-utf8`
- Croatian `croat`
- Czech (qwerty) `cz-lat2`
- Czech `cz-us-qwertz`
- Danish `dk`
- Danish (latin1) `dk-latin1`
- Devanagari (Inscript) `dev`
- Dutch `nl`
- Dvorak `dvorak`
- Estonian `et`
- Finnish (latin1) `fi-latin1`
- Finnish `fi`
- French (latin1) `fr-latin1`
- French (latin9) `fr-latin9`
- French (pc) `fr-pc`
- French Canadian `cf`
- French `fr`
- German (latin1 w/ no deadkeys) `de-latin1-nodeadkeys`
- German (latin1) `de-latin1`
- German `de`
- Greek `gr`
- Gujarati (Inscript) `guj`
- Hungarian (101 key) `hu101`
- Hungarian `hu`
- Icelandic `is-latin1`

- Irish `ie`
- Italian (IBM) `it-ibm`
- Italian (it2) `it2`
- Italian `it`
- Japanese `jp106`
- Korean `ko`
- Latin American `la-latin1`
- Macedonian `mk-utf`
- Norwegian `no`
- Polish `pl2`
- Portuguese `pt-latin1`
- Punjabi (Inscript) `gur`
- Romanian Cedilla `ro-cedilla`
- Romanian `ro`
- Romanian Standard Cedilla `ro-std-cedilla`
- Romanian Standard `ro-std`
- Russian `ru`
- Serbian (latin) `sr-latin`
- Serbian `sr-cy`
- Slovak (qwerty) `sk-qwerty`
- Slovenian `slovene`
- Spanish `es`
- Swedish `sv-latin1`
- Swiss French (latin1) `fr_CH-latin1`
- Swiss French `fr_CH`
- Swiss German (latin1) `sg-latin1`
- Swiss German `sg`
- Tajik `tj`
- Tamil (Inscript) `tml-inscript`
- Tamil (Typewriter) `tml-uni`
- Turkish `trq`
- U.S. English `us`
- U.S. International `us-acentos`
- Ukrainian `ua-utf`
- United Kingdom `uk`

groups

Within an *installation* section, the `groups` sub-section describes extra operating system groups to create as part of the machine image build process.

The definition of a `groups` section when using YAML is:

```
---
groups:
- # the list of groups goes here.
```

If you are using JSON:

```
"groups": [
    ...the list of groups goes here.
]
```

The valid keys to use within a group are:

- `name` (mandatory): a string providing the name of the group. The name cannot contain any spaces.
- `systemGroup` (optional): a boolean determining if the group is a system user.
- `groupId` (optional): an integer providing the unique Id of the group. This number must be greater than 1000. If the group is a system group, then this number must be greater than 201.

Examples

Basic Example

The following example describes groups to be created during the build. As no `groupId` is specified, the next available group Id numbers are used automatically during the build of the machine image.

If you are using YAML:

```
---
groups:
- name: nginx
- name: mongoDb
```

If you are using JSON:

```
{
  "groups": [
    {
      "name": "nginx"
    },
    {
      "name": "mongoDb"
    }
  ]
}
```

System Groups and Group Ids

This example shows how you can pre-determine the `groupId` of the group to be created as well as making the group a system group.

> **Warning:** A normal group's Id must be greater than 1000. If the group is a system group, then this Id can start at 201.

If you are using YAML:

```
---
groups:
- name: nginx
  groupId: 1033
- name: mongoDb
  systemGroup: true
  groupId: 245
```

If you are using JSON:

```
{
  "groups": [
    {
      "name": "nginx",
      "groupId": 1033
    },
    {
      "name": "mongoDb",
      "systemGroup": true,
      "groupId": 245
    }
  ]
}
```

internetSettings

Within an *installation*, the `internetSettings` sub-section describes information for the internet connections. If no information is provided, when an instance is provisioned from the machine image, the `basic` option is setup with one Ethernet card with dhcp configured.

If you are using JSON, the following is an example with multi-nics setup:

```
      {
"stack" : {
  "name" : "MyTemplate",
  "version" : "1.0",
  "installation" : {
    "internetSettings" : "configure",
    "nics" : [
      {
        "name" : "nic_1",
        "type" : "ETHERNET",
        "order" : 1,
        "autoConnect" : "true",
        "ipv4" : "dhcp",
```

```
        "ipv6" : "disabled"
      },
      {
        "name" : "nic_2",
        "type" : "ETHERNET",
        "order" : 2,
        "autoConnect" : "true",
        "ipv4" : "static",
        "ipv6" : "disabled",
        "ipAddresses" : [
          {
            "version" : 4,
            "address" : "10.0.0.111",
            "netmask" : "255.255.255.0",
            "gateway" : "10.0.0.1"
          }
        ]
      }
    ],
    "diskSize" : 12288,
    "swapSize" : 512
  },
  "os" : {
    "name" : "CentOS",
    "version" : "7",
    "arch" : "x86_64",
    "profile" : "Minimal"
  }
```

The valid keys to use within `internetSettings` are:

- basic: one Ethernet card with dhcp configured.
- ask: ask during installation.
- configure: nics to be configured in the template. If no nics definitions found in the json or yaml file, will get a bad request error.
- no value: considered as basic.

Sub-sections

The internetSettings sub-sections are:

nics

The following is a list of all the accepted values that can be used for the internet settings mode in an *installation* sub-section of the stack.

For example:

```
  "installation" : {
"internetSettings" : "configure",
"nics" : [
  {
    "name" : "nic_1",
    "type" : "ETHERNET",
```

```
    "order" : 1,
    "autoConnect" : "true",
    "ipv4" : "dhcp",
    "ipv6" : "disabled"
  },
  {
    "name" : "nic_2",
    "type" : "ETHERNET",
    "order" : 2,
    "autoConnect" : "true",
    "ipv4" : "static",
    "ipv6" : "disabled",
    "ipAddresses" : [
      {
        "version" : 4,
        "address" : "10.0.0.111",
        "netmask" : "255.255.255.0",
        "gateway" : "10.0.0.1"
      }
```

- **nics: list of nics objects, each nic has:**
  - name: connection name.
  - type: nic type only "Ethernet" supported for now.
  - order(optional): nic's order (1,2,3,...) if all nics do not have order their order in the list will be taken.
  - autoConnect: true/false. optional field default value (true).
  - ipv4: ipv4 method [dhcp/static/disabled]
  - ipv6: ipv6 method [dhcp/static/disabled]
  - **ipAddresses: list of ip addresses, each ipAddress has:**
    - version: IP version 4 / 6
    - address: the ip address
    - netmask: the network mask
    - gateway: the gateway

partitioning

Within an *installation* section, the partitioning sub-section allows you to describe an advanced partitioning table.

> **Warning:** Not all clouds support advanced partitioning. When building a machine image for an environment that does not support advanced partitioning, the build will fail with an appropriate error message.

The definition of a `partitioning` section when using YAML is:

```
---
partitioning:
  # the partitioning definition goes here.
```

If you are using JSON:

```
"partitioning": {
    ...the partitioning definition goes here.
}
```

The valid keys to use within partitioning are:

- `disks` (mandatory): an array of objects describing the physical disks that make up the advanced partitioning table. A disk may include up to four partitions, one of which can be of type extended that may hold one or more logical partitions. For more information, refer to the *disks* sub-section.
- `volumeGroups` (optional): an array objects describing any volume groups that make up the advanced partitioning table. If used, you must have one or more logical volumes using this volume group. For more information, refer to the *volumeGroups* sub-section.
- `logicalVolumes` (optional): an array of objects describing any logical volumes that make up the partitioning table. If used, you must have one or more volume groups that are used in creating the logical volume. For more information, refer to the *logical volumes* sub-section.

Sub-sections

The partitioning sub-sections are:

disks

Within a *partitioning* section, the `disks` sub-section describes all the physical disks for an advanced partitioning table. Each disk may be partitioned, i.e. the act of dividing the physical disk into logical sections with the goal to treat one physical disk drive as if it were multiple disks. These are called physical partitions. A disk may have a maximum of 4 physical partitions.

The definition of a `disks` section when using YAML is:

```
---
disks:
- # the list of disks goes here.
```

If you are using JSON:

```
"disks": [
    ...the list of disks goes here.
]
```

The valid keys to use within a disk are:

- `name` (mandatory): a string providing the name of the disk. Currently only the following values are valid: `sd` followed by a letter. The first disk must use the letter `a` (i.e. `sda`), the second disk `b` and so forth.
- `partitions` (optional): an array of objects describing the partitions for this disk. For more information, refer to the *partitions* sub-section. A maximum of 4 partitions is allowed.
- `size` (mandatory): an integer providing the size of the disk (in MB)
- `type` (mandatory): a string providing the disk type. The valid values are `MSDOS` or `LVM`

Sub-sections

The disks sub-sections are:

partitions

Within a *disks* section, the `partitions` sub-section describes all the partitions to create for the disk. Disk partitioning is the act of dividing the physical disk into logical sections with the goal to treat one physical disk drive as if it were multiple disks.

> **Warning:** A disk may have a maximum of 4 partitions.

The definition of a `partitions` section when using YAML is:

```
---
partitions:
- # the list of partitions goes here.
```

If you are using JSON:

```
"partitions": [
    ...the list of partitions goes here.
]
```

The valid keys to use within a partition are:

- `fstype` (mandatory): a string providing the filesystem type. See below for valid values.
- `grow` (optional): a boolean marking this partition as growable. When a partition is growable it will take any available space left on the disk after all the other partitions catered for. You can only have 1 growable partition in a disk.
- `label` (optional): a string providing a label for this partition
- `partitions` (optional): an array of objects `partition` describing any logical partitions this partition may contain. To use logical partitions, this partition must use the `Extended` filesystem type.
- `mountPoint` (optional): a string providing the mount point of the partition. If the `fstype` is NOT `lvm` then the mount point is mandatory.
- `number` (mandatory): an integer providing the partition number. Starting at 1
- `size` (mandatory): an integer providing the size of the partition. Note that the sum of all the partitions cannot be greater than the total disk size provided in the `disk`. The minimum size is 64MB.

Available Filesystem Types

The following are valid filesystem types used with the `fstype` key:

- `Extended`
- `ext2`
- `ext3`
- `ext4`
- `NTFS`
- `FAT16`
- `FAT32`
- `jfs`

- linux-swap
- lvm2
- unformated
- xfs

logical volumes

Within a *partitioning* section, the logicalVolumes sub-section describes the way a volume group should be partitioned.

The definition of a logicalVolumes section when using YAML is:

```
---
disks:
- # the list of logical volumes goes here.
```

If you are using JSON:

```
"disks": [
    ...the list of logical volumes goes here.
]
```

The valid keys to use within a logical volume are:

- fstype (mandatory): a string providing the filesystem type. See below for valid values.
- grow (optional): a boolean marking this volume (partition) as growable. When a volume is growable it will take any available space remaining in the volume group after all the other volumes catered for. You can only have 1 growable partition in the logical volume.
- label (optional): a string providing a label for this volume
- mountPoint (mandatory): a string providing the mount point of the volume.
- name (mandatory): a string providing the name of the volume.
- size (mandatory): an integer providing the size of the volume. Note that the sum of all the volumes cannot be greater than the total size provided in the *volumeGroups*.
- vg_name (mandatory): a string providing the name of the volume group this logical volume is using

Available Filesystem Types

The following are valid filesystem types used with the fstype key:

- Extended
- ext2
- ext3
- ext4
- NTFS
- FAT16
- FAT32

- jfs
- linux-swap
- lvm2
- unformated
- xfs

volumeGroups

Within a *partitioning* section, the volumeGroups sub-section describes a volume group in the partitioning table. A volume group creates a pool of disk space from a collection of disks or partitions. This volume group can then be partitioned via a *logical volumes*.

Warning: Only disks and partitions of type lvm and logical partitions can be added to a volume group.

The definition of a volumeGroups section when using YAML is:

```
---
disks:
- # the list of volume groups goes here.
```

If you are using JSON:

```
"disks": [
    ...the list of volume groups goes here.
]
```

The valid keys to use within a volumeGroup are:

- name (mandatory): a string providing the name of the volume group
- physicalVolumes (optional): an array of strings describing the names of the disks or partitions to include in this group. The sum of all the disks and partitions added will be the total size of this volume group. This logical pool of disk can then be partitioned via a logical volume. You cannot add a disk and one of its partitions into the same volume group, furthermore you cannot use a disk or partition more than once if 2 or more volume groups are declared.

Examples

Basic Example

The following example describes a partitioning table with one disk that has three partitions: the boot partition, a swap partition and a third partition called space.

If you are using YAML:

```
---
partitioning:
  disks:
  - name: sda
        type: msdos
        size: 12288
        partitions:
        - number: 1
          fstype: ext3
          size: 2048
          mountPoint: "/boot"
        - number: 2
          fstype: linux-swap
          size: 1024
        - number: 3
          fstype: ext3
          size: 9216
          label: space
          mountPoint: "/space"
```

If you are using JSON:

```
{
  "partitioning": {
    "disks": [
      {
        "name": "sda",
        "type": "msdos",
        "size": 12288,
        "partitions": [
          {
            "number": 1,
            "fstype": "ext3",
            "size": 2048,
            "mountPoint": "/boot"
          },
          {
            "number": 2,
            "fstype": "linux-swap",
            "size": 1024
          },
          {
            "number": 3,
            "fstype": "ext3",
            "size": 9216,
            "label": "space",
            "mountPoint": "/space"
          }
        ]
      }
    ]
  }
}
```

Using Growable Example

The same partitioning table as shown in *Basic Example* can be written slightly differently using the `grow` flag. A growable partition is partition that takes up the rest of the available disk space after the other partition sizes have been satisfied. In this case, we say that the "space" partition takes up the rest of the disk (rather than us having to calculate the space left after creating the first two partitions). We must specify though a size for the third partition (the minimum partition size is 64MB).

If you are using YAML:

```
---
partitioning:
  disks:
  - name: sda
        type: msdos
        size: 12288
        partitions:
        - number: 1
          fstype: ext3
          size: 2048
          mountPoint: "/boot"
        - number: 2
          fstype: linux-swap
          size: 1024
        - number: 3
          fstype: ext3
          size: 64
          grow: true
          label: space
          mountPoint: "/space"
```

If you are using JSON:

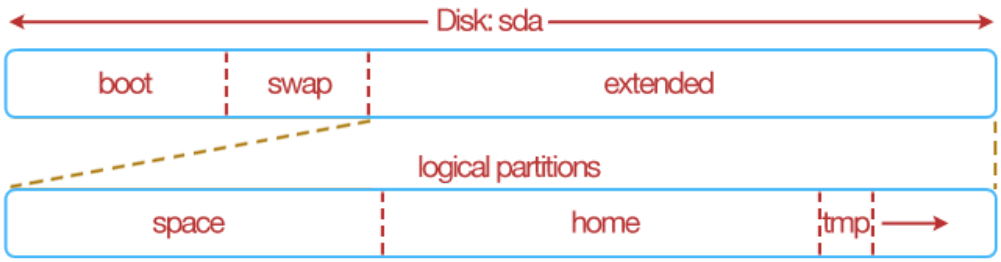
```
{
  "partitioning": {
    "disks": [
      {
        "name": "sda",
        "type": "msdos",
        "size": 12288,
        "partitions": [
          {
            "number": 1,
            "fstype": "ext3",
            "size": 2048,
            "mountPoint": "/boot"
          },
          {
            "number": 2,
            "fstype": "linux-swap",
            "size": 1024
```

```
            },
            {
              "number": 3,
              "fstype": "ext3",
              "size": 64,
              "grow": true,
              "label": "space",
              "mountPoint": "/space"
            }
          ]
        }
      ]
    }
}
```

Creating Logical Partitions Example

In this example the, logical partitions are created inside the `space` partition. The `space` partition now has the filesystem type `Extended`.

Warning: only one partition within a disk can have logical partitions. When a partition is `extended`, you cannot specify a mount point or a label. Logical partitions must start with number 5



If you are using YAML:

```
---
partitioning:
  disks:
  - name: sda
        type: msdos
        size: 12288
        partitions:
        - number: 1
          fstype: ext3
          size: 2048
          mountPoint: "/boot"
        - number: 2
          fstype: linux-swap
          size: 1024
        - number: 3
          fstype: Extended
          size: 9216
```

```
          partitions:
          - number: 5
                fstype: ext3
                size: 4098
                mountPoint: "/space"
                label: space
          - number: 6
                fstype: ext3
                size: 4098
                mountPoint: "/home"
                label: home
          - number: 7
                fstype: ext3
                size: 64
                mountPoint: "/tmp"
                label: tmp
                grow: true
```

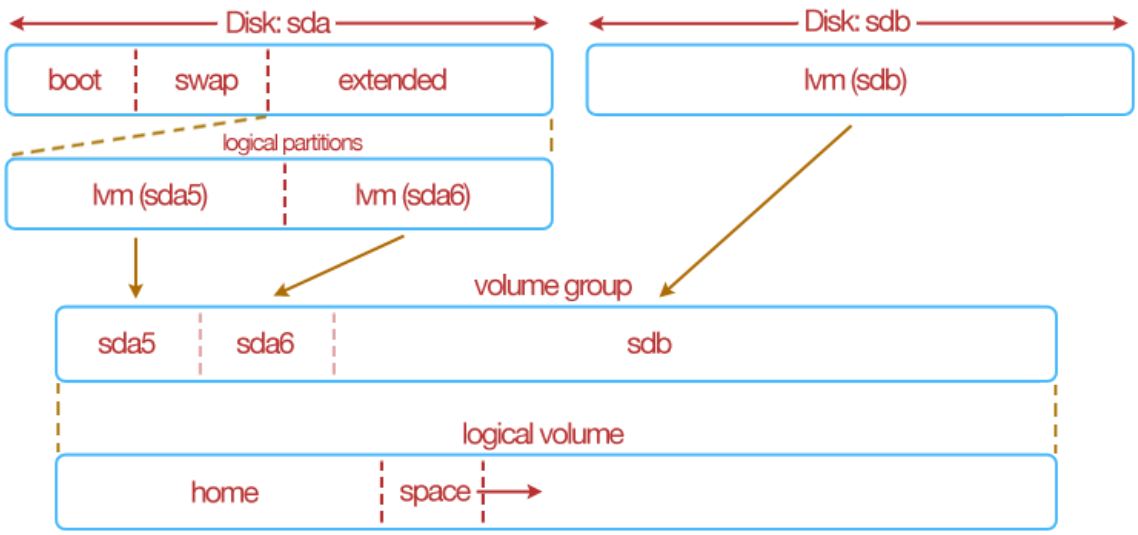
If you are using JSON:

```
{
  "partitioning": {
    "disks": [
      {
        "name": "sda",
        "type": "msdos",
        "size": 12288,
        "partitions": [
          {
            "number": 1,
            "fstype": "ext3",
            "size": 2048,
            "mountPoint": "/boot"
          },
          {
            "number": 2,
            "fstype": "linux-swap",
            "size": 1024
          },
          {
            "number": 3,
            "fstype": "Extended",
            "size": 9216,
            "partitions": [
              {
                "number": 5,
                "fstype": "ext3",
                "size": 4098,
                "mountPoint": "/space",
                "label": "space"
              },
              {
                "number": 6,
                "fstype": "ext3",
                "size": 4098,
                "mountPoint": "/home",
                "label": "home"
              },
```

```
                {
                  "number": 7,
                  "fstype": "ext3",
                  "size": 64,
                  "mountPoint": "/tmp",
                  "label": "tmp",
                  "grow": true
                }
              ]
            }
          ]
        }
      ]
    }
}
```

Volume Groups and Logical Volumes Example

The following example shows how disks and partitions that have lvm filesystem types can be regrouped together – volume group then re-partitioned differently – a logical volume.



If you are using YAML:

```
---
partitioning:
  disks:
  - name: sda
        type: msdos
        size: 12288
        partitions:
        - number: 1
          fstype: ext3
          mountPoint: "/boot"
          size: 1024
        - number: 2
```

```
          fstype: linux-swap
          size: 1024
        - number: 3
          fstype: extended
          size: 64
          grow: true
          partitions:
          - number: 5
                fstype: lvm2
                size: 5120
          - number: 6
                fstype: lvm2
                size: 5120
  - name: sdb
        type: lvm
        size: 122880
  volumeGroups:
  - name: grp1
        physicalVolumes:
        - name: sda5
        - name: sda6
        - name: sdb
  logicalVolumes:
  - name: vol1
        vg_name: grp1
        fstype: ext3
        mountPoint: "/home"
        size: 4098
  - name: vol2
        vg_name: grp1
        fstype: ext3
        mountPoint: "/space"
        size: 64
        grow: true
```

If you are using JSON:

```
{
  "partitioning": {
        "disks": [
          {
                "name": "sda",
                "type": "msdos",
                "size": 12288,
                "partitions": [
                  {
                        "number": 1,
                        "fstype": "ext3",
                        "mountPoint": "/boot",
                        "size": 1024
                  },
                  {
                        "number": 2,
                        "fstype": "linux-swap",
                        "size": 1024
                  },
                  {
                        "number": 3,
```

```
                        "fstype": "extended",
                        "size": 64,
                        "grow": true,
                        "partitions": [
                          {
                                "number": 5,
                                "fstype": "lvm2",
                                "size": 5120
                          },
                          {
                                "number": 6,
                                "fstype": "lvm2",
                                "size": 5120
                          }
                        ]
                  }
                ]
          },
          {
                "name": "sdb",
                "type": "lvm",
                "size": 122880
          }
        ],
        "volumeGroups": [
          {
                "name": "grp1",
                "physicalVolumes": [
                  {
                        "name": "sda5"
                  },
                  {
                        "name": "sda6"
                  },
                  {
                        "name": "sdb"
                  }
                ]
          }
        ],
        "logicalVolumes": [
          {
                "name": "vol1",
                "vg_name": "grp1",
                "fstype": "ext3",
                "mountPoint": "/home",
                "size": 4098
          },
          {
                "name": "vol2",
                "vg_name": "grp1",
                "fstype": "ext3",
                "mountPoint": "/space",
                "size": 64,
                "grow": true
          }
        ]
  }
```

```
}
```

rootUser

Within an *installation*, the `rootUser` sub-section describes information for the root user to be created as part of the machine image build. If no root user information is provided, when an instance is provisioned from the machine image, the root user password is prompted.

The definition of a `rootUser` section when using YAML is:

```
---
rootUser:
  # the root user definition goes here.
```

If you are using JSON:

```
"rootUser": {
    ...the root user definition goes here.
}
```

The valid keys to use within a rootUser are:

- disablePasswordLogin (optional): a boolean to determine whether to disable the ability for the root user to login into a running instance using the root password.
- password (optional): a string providing the root user password. A blank password "" is valid.
- setPassword (mandatory): a boolean to determine whether to preset a password during the build or prompt the user to add a password during first boot of the instance.
- sshKeys (optional): an array of objects providing one or more public ssh keys for the root user. For more information, refer to the *sshKeys* sub-section.

Sub-sections

The root user sub-sections are:

sshKeys

The `sshKeys` sub-section describes one or more public SSH keys that can be used for a particular user.

The definition of an `sshKeys` section when using YAML is:

```
---
sshKeys:
- # the list of ssh keys goes here.
```

If you are using JSON:

```
"sshKeys": [
    ...the list of ssh keys goes here.
]
```

The valid keys to use within a sshKey are:

- `name` (mandatory): a string providing the name of the public SSH key
- `publicKey` (mandatory): a string providing the public key content. A public key must begin with string `ssh-rsa` or `ssh-dss`

Example

This example shows how to provide an ssh key for the root user of the operating system.

If you are using YAML:

```
---
sshKeys:
- name: admin-public
  publicKey: ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAABIwAAAQEA6NF8iallvQVp22WDkTkyrtvp9eWW6A8YVr+kz4TjGYe7gHzIw+niNltGEFHzD8+v1I2YJ6
```

If you are using JSON:

```
{
  "sshKeys": [
    {
      "name": "admin-public",
      "publicKey": "ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAABIwAAAQEA6NF8iallvQVp22WDkTkyrtvp9eWW6A8YVr+kz4TjGYe7gHzIw+niNltGEFHzD8+v1I2YJ6
↪"
    }
  ]
}
```

Examples

Basic Example

The following example describes the basic root user information.

If you are using YAML:

```
---
rootUser:
  password: welcome-not-a-good-password
```

If you are using JSON:

```
{
  "rootUser": {
    "password": "welcome-not-a-good-password"
  }
}
```

Disabling Password Login

You can easily disable root password login as a security measure by doing the following:

If you are using YAML:

```
---
rootUser:
  password: welcome-not-a-good-password
  disablePasswordLogin: true
```

If you are using JSON:

```
{
  "rootUser": {
    "password": "welcome-not-a-good-password",
    "disablePasswordLogin": true
  }
}
```

timezone

The following is a list of all the accepted timezone values that can be used for the timezone key in an *installation* sub-section of the stack.

For example:

If you are using YAML:

```
---
installation:
  timezone: Europe/Paris
```

If you are using JSON:

```
{
  "installation": {
    "timezone": "Europe/Paris"
  }
}
```

Available Keyboard Values

- Africa/Abidjan
- Africa/Accra
- Africa/Addis_Ababa
- Africa/Algiers
- Africa/Asmara
- Africa/Bamako
- Africa/Bangui
- Africa/Banjul
- Africa/Bissau
- Africa/Blantyre

- Africa/Brazzaville
- Africa/Bujumbura
- Africa/Cairo
- Africa/Casablanca
- Africa/Ceuta
- Africa/Conakry
- Africa/Dakar
- Africa/Dar_es_Salaam
- Africa/Djibouti
- Africa/Douala
- Africa/El_Aaiun
- Africa/Freetown
- Africa/Gaborone
- Africa/Harare
- Africa/Johannesburg
- Africa/Kampala
- Africa/Khartoum
- Africa/Kigali
- Africa/Kinshasa
- Africa/Lagos
- Africa/Libreville
- Africa/Lome
- Africa/Luanda
- Africa/Lubumbashi
- Africa/Lusaka
- Africa/Malabo
- Africa/Maputo
- Africa/Maseru
- Africa/Mbabane
- Africa/Mogadishu
- Africa/Monrovia
- Africa/Nairobi
- Africa/Ndjamena
- Africa/Niamey
- Africa/Nouakchott
- Africa/Ouagadougou

- Africa/Porto-Novo
- Africa/Sao_Tome
- Africa/Tripoli
- Africa/Tunis
- Africa/Windhoek
- America/Adak
- America/Anchorage
- America/Anguilla
- America/Antigua
- America/Araguaina
- America/Argentina/Buenos_Aires
- America/Argentina/Catamarca
- America/Argentina/Cordoba
- America/Argentina/Jujuy
- America/Argentina/La_Rioja
- America/Argentina/Mendoza
- America/Argentina/Rio_Gallegos
- America/Argentina/Salta
- America/Argentina/San_Juan
- America/Argentina/San_Luis
- America/Argentina/Tucuman
- America/Argentina/Ushuaia
- America/Aruba
- America/Asuncion
- America/Atikokan
- America/Bahia
- America/Bahia_Banderas
- America/Barbados
- America/Belize
- America/Blanc-Sablon
- America/Boa_Vista
- America/Bogota
- America/Boise
- America/Cambridge_Bay
- America/Campo_Grande
- America/Cancun

- America/Caracas
- America/Cayenne
- America/Cayman
- America/Chicago
- America/Chihuahua
- America/Costa_Rica
- America/Cuiaba
- America/Curacao
- America/Danmarkshavn
- America/Dawson
- America/Dawson_Creek
- America/Denver
- America/Detroit
- America/Dominica
- America/Edmonton
- America/Eirunepe
- America/El_Salvador
- America/Fortaleza
- America/Glace_Bay
- America/Godthab
- America/Goose_Bay
- America/Grand_Turk
- America/Grenada
- America/Guadeloupe
- America/Guatemala
- America/Guayaquil
- America/Guyana
- America/Halifax
- America/Havana
- America/Hermosillo
- America/Indiana/Indianapolis
- America/Indiana/Knox
- America/Indiana/Marengo
- America/Indiana/Petersburg
- America/Indiana/Tell_City
- America/Indiana/Vevay

- America/Indiana/Vincennes
- America/Indiana/Winamac
- America/Inuvik
- America/Iqaluit
- America/Jamaica
- America/Juneau
- America/Kentucky/Louisville
- America/Kentucky/Monticello
- America/La_Paz
- America/Lima
- America/Los_Angeles
- America/Maceio
- America/Managua
- America/Manaus
- America/Marigot
- America/Martinique
- America/Matamoros
- America/Mazatlan
- America/Menominee
- America/Merida
- America/Mexico_City
- America/Miquelon
- America/Moncton
- America/Monterrey
- America/Montevideo
- America/Montreal
- America/Montserrat
- America/Nassau
- America/New_York
- America/Nipigon
- America/Nome
- America/Noronha
- America/North_Dakota/Beulah
- America/North_Dakota/Center
- America/North_Dakota/New_Salem
- America/Ojinaga

- America/Panama
- America/Pangnirtung
- America/Paramaribo
- America/Phoenix
- America/Port-au-Prince
- America/Port_of_Spain
- America/Porto_Velho
- America/Puerto_Rico
- America/Rainy_River
- America/Rankin_Inlet
- America/Recife
- America/Regina
- America/Resolute
- America/Rio_Branco
- America/Santa_Isabel
- America/Santarem
- America/Santiago
- America/Santo_Domingo
- America/Sao_Paulo
- America/Scoresbysund
- America/Shiprock
- America/St_Barthelemy
- America/St_Johns
- America/St_Kitts
- America/St_Lucia
- America/St_Thomas
- America/St_Vincent
- America/Swift_Current
- America/Tegucigalpa
- America/Thule
- America/Thunder_Bay
- America/Tijuana
- America/Toronto
- America/Tortola
- America/Vancouver
- America/Whitehorse

- America/Winnipeg
- America/Yakutat
- America/Yellowknife
- Antarctica/Casey
- Antarctica/Davis
- Antarctica/DumontDUrville
- Antarctica/Macquarie
- Antarctica/Mawson
- Antarctica/McMurdo
- Antarctica/Palmer
- Antarctica/Rothera
- Antarctica/South_Pole
- Antarctica/Syowa
- Antarctica/Vostok
- Arctic/Longyearbyen
- Asia/Aden
- Asia/Almaty
- Asia/Amman
- Asia/Anadyr
- Asia/Aqtau
- Asia/Aqtobe
- Asia/Ashgabat
- Asia/Baghdad
- Asia/Bahrain
- Asia/Baku
- Asia/Bangkok
- Asia/Beirut
- Asia/Bishkek
- Asia/Brunei
- Asia/Choibalsan
- Asia/Chongqing
- Asia/Colombo
- Asia/Damascus
- Asia/Dhaka
- Asia/Dili
- Asia/Dubai

- Asia/Dushanbe
- Asia/Gaza
- Asia/Harbin
- Asia/Ho_Chi_Minh
- Asia/Hong_Kong
- Asia/Hovd
- Asia/Irkutsk
- Asia/Jakarta
- Asia/Jayapura
- Asia/Jerusalem
- Asia/Kabul
- Asia/Kamchatka
- Asia/Karachi
- Asia/Kashgar
- Asia/Kathmandu
- Asia/Kolkata
- Asia/Krasnoyarsk
- Asia/Kuala_Lumpur
- Asia/Kuching
- Asia/Kuwait
- Asia/Macau
- Asia/Magadan
- Asia/Makassar
- Asia/Manila
- Asia/Muscat
- Asia/Nicosia
- Asia/Novokuznetsk
- Asia/Novosibirsk
- Asia/Omsk
- Asia/Oral
- Asia/Phnom_Penh
- Asia/Pontianak
- Asia/Pyongyang
- Asia/Qatar
- Asia/Qyzylorda
- Asia/Rangoon

- Asia/Riyadh
- Asia/Sakhalin
- Asia/Samarkand
- Asia/Seoul
- Asia/Shanghai
- Asia/Singapore
- Asia/Taipei
- Asia/Tashkent
- Asia/Tbilisi
- Asia/Tehran
- Asia/Thimphu
- Asia/Tokyo
- Asia/Ulaanbaatar
- Asia/Urumqi
- Asia/Vientiane
- Asia/Vladivostok
- Asia/Yakutsk
- Asia/Yekaterinburg
- Asia/Yerevan
- Atlantic/Azores
- Atlantic/Bermuda
- Atlantic/Canary
- Atlantic/Cape_Verde
- Atlantic/Faroe
- Atlantic/Madeira
- Atlantic/Reykjavik
- Atlantic/South_Georgia
- Atlantic/St_Helena
- Atlantic/Stanley
- Australia/Adelaide
- Australia/Brisbane
- Australia/Broken_Hill
- Australia/Currie
- Australia/Darwin
- Australia/Eucla
- Australia/Hobart

- Australia/Lindeman
- Australia/Lord_Howe
- Australia/Melbourne
- Australia/Perth
- Australia/Sydney
- Europe/Amsterdam
- Europe/Athens
- Europe/Belgrade
- Europe/Berlin
- Europe/Bratislava
- Europe/Brussels
- Europe/Bucharest
- Europe/Budapest
- Europe/Chisinau
- Europe/Copenhagen
- Europe/Dublin
- Europe/Gibraltar
- Europe/Guernsey
- Europe/Helsinki
- Europe/Isle_of_Man
- Europe/Istanbul
- Europe/Jersey
- Europe/Kaliningrad
- Europe/Kiev
- Europe/Lisbon
- Europe/Ljubljana
- Europe/London
- Europe/Luxembourg
- Europe/Madrid
- Europe/Malta
- Europe/Mariehamn
- Europe/Minsk
- Europe/Monaco
- Europe/Moscow
- Europe/Oslo
- Europe/Paris

- Europe/Podgorica
- Europe/Prague
- Europe/Riga
- Europe/Rome
- Europe/Samara
- Europe/San_Marino
- Europe/Sarajevo
- Europe/Simferopol
- Europe/Skopje
- Europe/Sofia
- Europe/Stockholm
- Europe/Tallinn
- Europe/Tirane
- Europe/Uzhgorod
- Europe/Vaduz
- Europe/Vatican
- Europe/Vienna
- Europe/Vilnius
- Europe/Volgograd
- Europe/Warsaw
- Europe/Zagreb
- Europe/Zaporozhye
- Europe/Zurich
- Indian/Antananarivo
- Indian/Chagos
- Indian/Christmas
- Indian/Cocos
- Indian/Comoro
- Indian/Kerguelen
- Indian/Mahe
- Indian/Maldives
- Indian/Mauritius
- Indian/Mayotte
- Indian/Reunion
- Pacific/Apia
- Pacific/Auckland

- Pacific/Chatham
- Pacific/Chuuk
- Pacific/Easter
- Pacific/Efate
- Pacific/Enderbury
- Pacific/Fakaofo
- Pacific/Fiji
- Pacific/Funafuti
- Pacific/Galapagos
- Pacific/Gambier
- Pacific/Guadalcanal
- Pacific/Guam
- Pacific/Honolulu
- Pacific/Johnston
- Pacific/Kiritimati
- Pacific/Kosrae
- Pacific/Kwajalein
- Pacific/Majuro
- Pacific/Marquesas
- Pacific/Midway
- Pacific/Nauru
- Pacific/Niue
- Pacific/Norfolk
- Pacific/Noumea
- Pacific/Pago_Pago
- Pacific/Palau
- Pacific/Pitcairn
- Pacific/Pohnpei
- Pacific/Port_Moresby
- Pacific/Rarotonga
- Pacific/Saipan
- Pacific/Tahiti
- Pacific/Tarawa
- Pacific/Tongatapu
- Pacific/Wake
- Pacific/Wallis

users

Within an *installation* section, the `users` sub-section describes extra operating system users to create as part of the machine image build process.

The definition of a `users` section when using YAML is:

```
---
users:
- # the list of users goes here.
```

If you are using JSON:

```
"users": [
    ...the list of users goes here.
]
```

The valid keys to use within a user are:

- `fullName` (mandatory): a string providing the full name of the user. The same value as `name` can be used.
- `homeDir` (mandatory): a string providing the home directory of the user. Recommended default: `/home/username` where username is the same value as `name`
- `name` (mandatory): a string providing the name of the user. The name cannot contain any spaces.
- `password` (optional): a string providing the user password.
- `primaryGroup` (optional): a string providing the user's primary group. If no primary group is given, then the primary group is the same as name.
- `shell` (mandatory): a string providing the default shell environment for the user. Recommended default is `/bin/bash`.
- `secondaryGroups` (optional): a string providing one or more group names separated by a comma (,).
- `systemUser` (optional): a boolean determining if the user is a system user.
- `userId` (optional): an integer providing the unique Id of the user. This number must be greater than 1000. If the user is a system user, then this number must be greater than 201.

Examples

Basic Example

The following example provides the minimal information to create users during a build. As no `userId` is specified, the next available user Id numbers are used automatically during the build of the machine image. Furthermore, as no primary group is provided, the primary group will have the same name as the user name.

If you are using YAML:

```
---
users:
- name: joris
  fullName: joris
  homeDir: "/home/joris"
  shell: "/bin/bash"
- name: yann
  fullName: yann dorcet
```

```
  homeDir: "/home/ydorcet"
  shell: "/bin/bash"
```

If you are using JSON:

```
{
  "users": [
    {
      "name": "joris",
      "fullName": "joris",
      "homeDir": "/home/joris",
      "shell": "/bin/bash"
    },
    {
      "name": "yann",
      "fullName": "yann dorcet",
      "homeDir": "/home/ydorcet",
      "shell": "/bin/bash"
    }
  ]
}
```

More Complex Example

This example shows how you can provide group information, set a user Id and make a user a system user.

If you are using YAML:

```
---
users:
- name: joris
  fullName: joris
  userId: 2222
  primaryGroup: joris
  secondaryGroups: dev,france
  homeDir: "/home/joris"
  shell: "/bin/bash"
- name: yann
  fullName: yann dorcet
  systemUser: true
  userId: 400
  primaryGroup: yann
  secondaryGroups: admin,dev,france
  homeDir: "/home/ydorcet"
  shell: "/sbin/nologin"
```

If you are using JSON:

```
{
  "users": [
    {
      "name": "joris",
      "fullName": "joris",
      "userId": 2222,
      "primaryGroup": "joris",
      "secondaryGroups": "dev,france",
```

```
      "homeDir": "/home/joris",
      "shell": "/bin/bash"
    },
    {
      "name": "yann",
      "fullName": "yann dorcet",
      "systemUser": true,
      "userId": 400,
      "primaryGroup": "yann",
      "secondaryGroups": "admin,dev,france",
      "homeDir": "/home/ydorcet",
      "shell": "/sbin/nologin"
    }
  ]
}
```

Warning: By setting `/sbin/nologin` the user will not be able to log in via the machine's console.

selinux

The following is a list of all the accepted values that can be used for the SELinux mode in an *installation* sub-section of the stack.

For example:

```
{
  "installation": {
    "seLinuxMode" : "permissive"
  }
}
```

Accepted Values

- `disabled`
- `permissive`
- `enforcing`

Example

The following example sets the timezone, disk size, swap size, kernel parameters and automatically accepts all the licenses on the end-user's behalf (license information not displayed on boot).

Note: By default without any installation information specified, the internet settings is set to `basic`; kernel parameters `rhgb` and `quiet` are set and display licenses is set to `true`.

If you are using YAML:

```
---
installation:
  timezone: Europe/London
  internetSettings: basic
  kernelParams:
  - rhgb
  - quiet
  diskSize: 12288
  swapSize: 512
  displayLicenses: false
```

If you are using JSON:

```
{
  "installation": {
    "timezone": "Europe/London",
    "internetSettings": "basic",
    "kernelParams": [
      "rhgb",
      "quiet"
    ],
    "diskSize": 12288,
    "swapSize": 512,
    "displayLicenses": false
  }
}
```

config

Within a `stack`, the `config` sub-section describes boot scripts to be added to the template and executed as part of the boot sequence of an instance.

The definition of a `config` section when using YAML is:

```
---
config:
- # the list of configuration file definitions goes here.
```

If you are using JSON:

```
"config": [
    ...the list of configuration file definitions goes here.
]
```

The valid keys to use within a config are:

- `frequency` (mandatory): a string to determine when the boot script should be executed. There are only two valid values, firstboot to specify the script should only be executed once on the first time the instance is booted; everyboot to specify the script should be executed every time the instance is booted.
- `name` (mandatory): a string providing the name of the boot script.
- `source` (mandatory): a string providing the location of the boot script. This can be a filesystem pathname (relative or absolute) or an URL.
- `type` (mandatory): a string providing the script type. For the moment the only valid value is bootscript
- `order` (optional): an integer providing the boot order.

Example

The following example shows the declaration of two boot scripts, one to be execute only once the first time the machine is instantiated; the second to be executed every time the instance is rebooted.

If you are using YAML:

```
---
config:
- name: configure-mysql.sh
  source: "/home/joris/demo/configure-mysql.sh"
  type: bootscript
  frequency: firstboot
- name: check-stats.sh
  source: http://downloads.mysite.com/config/check-stats.sh
  type: bootscript
  frequency: everyboot
  order: '1'
```

If you are using JSON:

```
{
  "config": [
    {
      "name": "configure-mysql.sh",
      "source": "/home/joris/demo/configure-mysql.sh",
      "type": "bootscript",
      "frequency": "firstboot"
    },
    {
      "name": "check-stats.sh",
      "source": "http://downloads.mysite.com/config/check-stats.sh",
      "type": "bootscript",
      "frequency": "everyboot",
      "order": "1"
    }
  ]
}
```

Example

The following example shows a simple `stack` definition, when using YAML:

```
---
stack:
  name: CentOS Base Template
  version: '6.4'
  description: This is a CentOS core template.
  os:
        name: CentOS
        version: '6.4'
        arch: x86_64
        profile: Minimal
```

If you are using JSON:

```
{
  "stack": {
    "name": "CentOS Base Template",
    "version": "6.4",
    "description": "This is a CentOS core template.",
    "os": {
      "name": "CentOS",
      "version": "6.4",
      "arch": "x86_64",
      "profile": "Minimal"
    }
  }
}
```

Builders

Within a template, the `builders` section is an array of objects, describing the list of machine images to build (and where possible publish). For example if you wished to build an AMI image for Amazon EC2 and another for Microsoft Azure, you would specify a builder for each.

The information may include H/W requirements, authentication information (known as a cloud account) or where to upload and register the machine image after the build is complete.

Please refer to the specific machine image format for the mandatory and optional attributes.

A Word on Cloud Accounts

For "cloud" machine images, for example Amazon EC2, Azure CloudStack, OpenStack, Flexiant and Eucalyptus, the `builder` requires account information to the cloud environment. Information from the builder is used to correctly generate the machine image (for example AMI images for Amazon EC2 requires to have certain certificates embedded into the machine image) and to upload and register the machine image into the correct region, zone or datacenter.

The cloud account information can be part of the builder section, however as this includes sensitive information, hammr provides other mechanisms to include this information in the builder section. A safer way is to store this information in a separate file (JSON or YAML) and create the cloud account using `account create`; then reference the account `name` in the builder.

Please refer to the specific machine image format for the cloud account options and examples.

Cloud Targets

Abiquo

Default builder type: `Abiquo`

Require Cloud Account: Yes

www.abiquo.com

The `Abiquo` builder provides information for building and publishing the machine image for the Abiquo cloud platform. The Abiquo builder requires cloud account information to upload and register the machine image to the Abiquo platform. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The Abiquo builder section has the following definition when using YAML:

```
---
builders:
- type: Abiquo
  account:
    type: Abiquo
```

If you are using JSON:

```
{
  "builders": [
        {
          "type": "Abiquo",
          ...the rest of the definition goes here.
        }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Abiquo: `Abiquo`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. The followi**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
  - `hwType` (optional): an integer providing the hardware type for the machine image. This is the VMware hardware type: 4 (ESXi>3.x), 7 (ESXi>4.x) or 9 (ESXi>5.x)
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Abiquo, `Abiquo`. To get the available builder type, please refer to *format*
- `account` (mandatory): an object providing the abiquo cloud account information required to publish the built machine image.
- `category` (mandatory): a string providing the category this machine image. The category name must already be present in the abiquo platform.
- `datacenter` (mandatory): a string providing the datacenter name. The datacenter must already be present in the abiquo platform, and the cloud account must have access to the datacenter.
- `description` (mandatory): a string providing the description of what the machine image does.
- `enterprise` (mandatory): a string providing the enterprise resource name where to publish the machine image to. The enterprise resource must already exists in the abiquo platform, and the cloud account must have access to the enterprise resource.
- `productName` (mandatory): a string providing the name to be displayed for machine image. The name cannot exceed 32 characters

Abiquo Cloud Account

Key: `account`

Used to authenticate the abiquo platform.

The Abiquo cloud account has the following valid keys:

- `type` (mandatory): a string providing the cloud account type. Default platform type for Abiquo: `Abiquo`. To get the available platform type, please refer to *platform*
- `file` (optional): a string providing the location of the account information. This can be a pathname (relative or absolute) or an URL.
- `hostname` (mandatory): a string providing the hostname or IP address where the abiquo cloud platform is running
- `name` (mandatory): a string providing the name of the cloud account. This name can be used in a builder section to reference the rest of the cloud account information.
- `password` (mandatory): a string providing the password to use to authenticate
- `username` (mandatory): a string providing the username to use to authenticate

---

Note: In the case where `name` or `file` is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Examples

Basic Example

The following example shows an abiquo builder with all the information to build and publish a machine image to the Abiquo Cloud platform.

If you are using YAML:

```
---
builders:
- type: Abiquo
  account:
    type: Abiquo
    name: My Abiquo Account
    hostname: test.abiquo.com
    username: myLogin
    password: myPassWD
  hardwareSettings:
    memory: 1024
  installation:
    diskSize: 2000
  enterprise: UShareSoft
  datacenter: London
  productName: CentOS Core
  category: OS
  description: CentOS Core template.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Abiquo",
      "account": {
        "type": "Abiquo",
        "name": "My Abiquo Account",
        "hostname": "test.abiquo.com",
        "username": "myLogin",
        "password": "myPassWD"
      },
      "hardwareSettings": {
        "memory": 1024
      },
      "installation": {
        "diskSize": 2000
      },
      "enterprise": "UShareSoft",
      "datacenter": "London",
      "productName": "CentOS Core",
      "category": "OS",
      "description": "CentOS Core template."
    }
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a YAML file `abiquo-account.yml`.

```
---
account:
    type: Abiquo
```

```
    name: My Abiquo Account
    hostname: test.abiquo.com
    username: myLogin
    password: myPassWD
```

If you are using JSON, create a JSON file `abiquo-account.json`:

```
{
  "accounts": [
    {
      "type": "Abiquo",
      "name": "My Abiquo Account"
      "hostname": "test.abiquo.com",
      "username": "myLogin",
      "password": "myPassWD"
    }
  ]
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: Abiquo
  account:
    file: "/home/joris/accounts/abiquo-account.yml"
  hardwareSettings:
    memory: 1024
  installation:
    diskSize: 2000
  enterprise: UShareSoft
  datacenter: London
  productName: CentOS Core
  category: OS
  description: CentOS Core template.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Abiquo",
      "account": {
        "file": "/home/joris/accounts/abiquo-account.json"
      },
      "hardwareSettings": {
        "memory": 1024
      },
      "installation": {
        "diskSize": 2000
      },
      "enterprise": "UShareSoft",
      "datacenter": "London",
      "productName": "CentOS Core",
      "category": "OS",
```

```
      "description": "CentOS Core template."
    }
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: Abiquo
  account:
    name: My Abiquo Account
  hardwareSettings:
    memory: 1024
  installation:
    diskSize: 2000
  enterprise: UShareSoft
  datacenter: London
  productName: CentOS Core
  category: OS
  description: CentOS Core template.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Abiquo",
      "account": {
        "name": "My Abiquo Account"
      },
      "hardwareSettings": {
        "memory": 1024
      },
      "installation": {
        "diskSize": 2000
      },
      "enterprise": "UShareSoft",
      "datacenter": "London",
      "productName": "CentOS Core",
      "category": "OS",
      "description": "CentOS Core template."
    }
  ]
}
```

Amazon EC2

Default builder type: `Amazon AWS`

Require Cloud Account: Yes

aws.amazon.com

The Amazon builder provides information for building and publishing machine images for Amazon EC2. The Amazon builder requires cloud account information to upload and register the machine image to AWS (Amazon Web Services)

public cloud. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The Amazon builder section has the following definition when using YAML:

```
---
builders:
- type: Amazon AWS
  # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Amazon AWS",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Amazon: `Amazon AWS`. To get the available builder type, please refer to *format*
- `disableRootLogin` (optional): a boolean flag to determine if root login access should be disabled for any instance provisioned from the machine image.
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. As EBS-backed machine image is created, the maximum disk size is 1TB.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Amazon: `Amazon AWS`. To get the available builder type, please refer to *format*
- `account` (mandatory): an object providing the AWS cloud account information required to publish the built machine image.

- `region` (mandatory): a string providing the region where to publish the machine image. See below for valid regions.
- `bucket` (mandatory): a string providing the bucket name where to store the machine image. Bucket names are global to everyone, so you must choose a unique bucket name that does not already exist (or belongs to you). A bucket name cannot include spaces. Note, that if the bucket exists already in one region (for example Europe) and you wish to upload the same machine image to another region, then you must provide a new bucket name.

Valid Regions

The following regions are supported:

- `ap-northeast-1`: Asia Pacific (Tokyo) Region
- `ap-southeast-1`: Asia Pacific (Singapore) Region
- `eu-west-1`: EU (Ireland) Region
- `sa-east-1`: South America (Sao Paulo) Region
- `us-east-1`: US East (North Virginia) Region
- `us-west-1`: US West (North california) Region
- `us-west-2`: US West (Oregon) Region

Amazon Cloud Account

Key: `account`

Used to authenticate to AWS.

The Amazon cloud account has the following valid keys:

- `type` (mandatory): a string providing the cloud account type. Default platform type for Amazon: `Amazon`. To get the available platform type, please refer to *platform*
- `name` (mandatory): a string providing the name of the cloud account. This name can be used in a builder section to reference the rest of the cloud account information.
- `accountNumber` (mandatory): A string providing your AWS account number. This number can be found at the top right hand side of the Account > Security Credentials page after signing into Amazon Web Services
- `accessKeyId` (mandatory): A string providing your AWS access key id. To get your access key, sign into AWS (aws.amazon.com), click on Security Credentials > Access Credentials > Access Keys. Your access key id should be displayed, otherwise create a new one. Note, for security purposes, we recommend you change your access keys every 90 days
- `secretAccessKeyId` (mandatory): A string providing you AWS secret access key. To get your secret access key, sign into AWS (aws.amazon.com), click on Security Credentials > Access Credentials > Access Keys. Click on the Show button to reveal your secret key
- `x509Cert` (mandatory): A string providing the pathname or URL where to retrieve the X.509 certificate public key. To create a X.509 certificate, sign into AWS (aws.amazon.com), click on Security Credentials > Access Credentials > X.509 Certificates. Download the X.509 certificate or create a new one. This should be a (.pem) file.
- `x509PrivateKey` (mandatory): A string providing the pathname or URL where to retrieve the X.509 certificate private key. This private key is provided during the X.509 creation process. AWS does not store this private key, so you must download it and store it during this creation process. To create a X.509 certificate, sign into

AWS (aws.amazon.com), click on Security Credentials > Access Credentials > X.509 Certificates and create a new certificate. Download and save the Private Key. This should be a (.pem) file

- `file` (optional): a string providing the location of the account information. This can be a pathname (relative or absolute) or an URL.

Note: In the case where `name` or `file` is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Example

The following example shows an amazon builder with all the information to build and publish a machine image to Amazon EC2.

If you are using YAML:

```
---
builders:
- type: Amazon AWS
  account:
    type: Amazon
    name: My AWS account
    accountNumber: 11111-111111-1111
    accessKeyId: myaccessKeyid
    secretAccessKeyId: mysecretaccesskeyid
    x509Cert: "/path/to/aws.cert.pem"
    x509PrivateKey: "/path/to/aws.key.pem"
  installation:
    diskSize: 10240
  region: eu-central-1
  bucket: testsohammr
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Amazon AWS",
      "account": {
        "type": "Amazon",
        "name": "My AWS account",
        "accountNumber": "11111-111111-1111",
        "accessKeyId": "myaccessKeyid",
        "secretAccessKeyId": "mysecretaccesskeyid",
        "x509Cert": "/path/to/aws.cert.pem",
        "x509PrivateKey": "/path/to/aws.key.pem"
      },
      "installation": {
        "diskSize": 10240
      },
      "region": "eu-central-1",
      "bucket": "testsohammr"
    }
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a YAML file `aws-account.yml`.

```
---
accounts:
- type: Amazon
  accountNumber: 11111-111111-1111
  name: My AWS account
  accessKeyId: myaccessKeyid
  secretAccessKeyId: mysecretaccesskeyid
  x509Cert: "/path/to/aws.cert.pem"
  x509PrivateKey: "/path/to/aws.key.pem"
```

If you are using JSON, create a JSON file `aws-account.json`:

```
{
  "accounts": [
    {
      "type": "Amazon",
      "accountNumber": "11111-111111-1111",
      "name": "My AWS account",
      "accessKeyId": "myaccessKeyid",
      "secretAccessKeyId": "mysecretaccesskeyid",
      "x509Cert": "/path/to/aws.cert.pem",
      "x509PrivateKey": "/path/to/aws.key.pem"
    }
  ]
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: Amazon AWS
  account:
    file: "/path/to/aws-account.yml"
  installation:
    diskSize: 10240
  region: eu-central-1
  bucket: test-so-hammr
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Amazon AWS",
      "account": {
            "file": "/path/to/aws-account.json"
      },
      "installation": {
        "diskSize": 10240
      },
```

```
      "region": "eu-central-1",
      "bucket": "test-so-hammr"
    }
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: Amazon AWS
  account:
    name: My AWS Account
  installation:
    diskSize: 10240
  region: eu-central-1
  bucket: test-so-hammr
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Amazon AWS",
      "account": {
            "name": "My AWS Account"
      },
      "installation": {
        "diskSize": 10240
      },
      "region": "eu-central-1",
      "bucket": "test-so-hammr"
    }
  ]
}
```

CloudStack

Default builder type: `CloudStack VMware (OVA)`, `CloudStack KVM (QCOW2)` or `CloudStack Citrix Xen (VHD)`

Require Cloud Account: Yes

cloudstack.apache.org

The CloudStack builder provides information for building and publishing the machine image to the CloudStack cloud platform. This builder supports KVM (`CloudStack KVM (QCOW2)`), Xen (`CloudStack Citrix Xen (VHD)`) or VMware (`CloudStack VMware (OVA)`) based images for CloudStack. These builder types are the default names provided by UForge AppCenter.

Note: These builder type names can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The CloudStack builder requires cloud account information to upload and register the machine image to the CloudStack platform.

The CloudStack builder section has the following definition when using YAML:

```
---
builders:
- type: CloudStack KVM (QCOW2)
  # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "CloudStack KVM (QCOW2)",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for CloudStack: `CloudStack VMware (OVA)`, `CloudStack KVM (QCOW2)` or `CloudStack Citrix Xen (VHD)`. To get the available builder type, please refer to *format*
- `hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. If an OVF machine image is being built, then the hardware settings are mandatory. The following valid keys for hardware settings are:
- `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
- `installation` (optional): an object providing low-level installation or first boot options. These override any installation options in the *Stack* section. The following valid keys for installation are: * `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for CloudStack: `CloudStack VMware (OVA)`, `CloudStack KVM (QCOW2)` or `CloudStack Citrix Xen (VHD)`. To get the available builder type, please refer to *format*
- `account` (mandatory): an object providing the CloudStack cloud account information required to publish the built machine image.

- `imageName` (mandatory): a string providing the displayed name of the machine image.
- `zone` (mandatory): a string providing the zone to publish the machine image
- `publicImage` (optional): a boolean flag to determine in the machine image is to be public
- `featured` (optional): a boolean flag to determine in the machine image is to be "featured"

CloudStack Cloud Account

Key: `account` Used to authenticate the CloudStack platform.

The CloudStack cloud account has the following valid keys:

- `type` (mandatory): a string providing the cloud account type. Default platform type for CloudStack is `CloudStack`. To get the available platform type, please refer to *platform*
- `name` (mandatory): a string providing the name of the cloud account. This name can be used in a builder section to reference the rest of the cloud account information.
- `publicApiKey` (mandatory): a string providing your public API key. If you do not have a public/secret key pair, please refer to the CloudStack documentation to generate them, or contact your cloud administrator
- `secretApiKey` (mandatory): a string providing your secret API key. If you do not have a public/secret key pair, please refer to the CloudStack documentation to generate them, or contact your cloud administrator
- `endpointUrl` (mandatory): a string providing the API URL endpoint of the cloudstack management console to upload the machine image to. For example: http://cloudstackhostname:8080/client/api
- `file` (optional): a string providing the location of the account information. This can be a pathname (relative or absolute) or an URL.

Note: In the case where `name` or `file` is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Example

The following example shows a CloudStack builder with all the information to build and publish a machine image to CloudStack.

If you are using YAML:

```
---
builders:
- type: CloudStack KVM (QCOW2)
  account:
    type: CloudStack
    name: My CloudStack account
    publicApiKey: mypublicapikey
    secretApiKey: mysecretapiKey
    endpointUrl: myendpointurl
  imageName: CentOS Core
  zone: zone1
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "CloudStack KVM (QCOW2)",
      "account": {
        "type": "CloudStack",
        "name": "My CloudStack account",
        "publicApiKey": "mypublicapikey",
        "secretApiKey": "mysecretapiKey",
        "endpointUrl": "myendpointurl"
      },
      "imageName": "CentOS Core",
      "zone": "zone1"
    }
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a YAML file `cloudstack-account.yml`.

```
---
accounts:
- type: CloudStack
  name: My CloudStack account
  publicApiKey: mypublicapikey
  secretApiKey: mysecretapiKey
  endpointUrl: myendpointurl
```

If you are using JSON, create a JSON file `cloudstack-account.json`:

```
{
  "accounts": [
    {
      "type": "CloudStack",
      "name": "My CloudStack account",
      "publicApiKey": "mypublicapikey",
      "secretApiKey": "mysecretapiKey",
      "endpointUrl": "myendpointurl"
    }
  ]
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: CloudStack KVM (QCOW2)
  account:
    file: "/path/to/cloudstack-account.yml"
```

```
  imageName: CentOS Core
  zone: zone1
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "CloudStack KVM (QCOW2)",
      "account": {
        "file": "/path/to/cloudstack-account.json"
      },
      "imageName": "CentOS Core",
      "zone": "zone1"
    }
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: CloudStack KVM (QCOW2)
  account:
    name: My CloudStack Account
  imageName: CentOS Core
  zone: zone1
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "CloudStack KVM (QCOW2)",
      "account": {
        "name": "My CloudStack Account"
      },
      "imageName": "CentOS Core",
      "zone": "zone1"
    }
  ]
}
```

Eucalyptus

Default builder type: `Eucalyptus KVM` or `Eucalyptus XEN`

Require Cloud Account: Yes

www.eucalyptus.com

The Eucalyptus builder provides information for building and publishing the machine image for Eucalyptus. This builder supports KVM (`Eucalyptus KVM`) and Xen (`Eucalyptus XEN`) based images for Eucalyptus. These builder types are the default names provided by UForge AppCenter.

Note: These builder type names can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The Eucalyptus builder requires cloud account information to upload and register the machine image to an Eucalyptus cloud platform.

The Eucalyptus builder section has the following definition when using YAML:

```
---
builders:
- type: Eucalyptus KVM
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Eucalyptus KVM",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Eucalyptus: `Eucalyptus KVM` or `Eucalyptus XEN`. To get the available builder type, please refer to *format*
- `account` (mandatory): an object providing the Eucalyptus cloud account information required to publish the built machine image.
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.
- `disableRootLogin` (optional): a boolean flag to determine if root login access should be disabled for any instance provisioned from the machine image.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Eucalyptus: `Eucalyptus KVM` or `Eucalyptus XEN`. To get the available builder type, please refer to *format*
- `account` (mandatory): an object providing the Eucalyptus cloud account information required to publish the built machine image.
- `bucket` (mandatory): a string providing the bucket where to store the machine image. Note, bucket names are global to everyone, so you must choose a unique bucket name that is not used by another user. The bucket name should not include spaces.
- `description` (mandatory): a string providing a description of what the machine image does. The description of the machine image is displayed in the console. The description can only be up to 255 characters long. Descriptions longer than 255 characters will be truncated.
- `imageName` (mandatory): a string providing the displayed name for the machine image.
- `kernelId` (optional): a string providing the kernel Id when booting an instance from the machine image. Note that the kernel id must be already present on the cloud environment. If a kernel Id is not specified, then the default kernel Id registered on the cloud platform will be used.
- `ramdisk` (optional): a string providing the ramdisk Id when booting an instance from the machine image. Note that the ramdisk Id must be already present on the cloud environment. If a ramdisk Id is not specified, then the default ramdisk Id registered on the cloud platform will be used.

Eucalyptus Cloud Account

Key: `account` Used to authenticate to Eucalyptus.

The Eucalyptus cloud account has the following valid keys:

- `type` (mandatory): a string providing the cloud account type. Default platform type for Eucalyptus is `Eucalyptus`. To get the available platform type, please refer to *platform*
- `accountNumber` (mandatory): a string providing the User ID or Eucalyptus account number of the user who is bundling the image. This value can be found in the eucarc file.
- `cloudCert` (mandatory): a string providing the location of the cloud certificate. This may be a path or URL. To get the cloud certificate, login into your Eucalyptus admin console (for example https://myserver.domain.com:8443). Go to the Credentials ZIP-file and click on the button Download credentials. Unzip this file, you should find the certificate with the name cloud-cert.pem
- `file` (optional): a string providing the location of the account information. This can be a pathname (relative or absolute) or an URL.
- `endpoint` (mandatory): a string providing the URL of the Eucalyptus Walrus server. To get the walrus server information, login into your Eucalyptus admin console and click on the Configuration tab
- `name`: (mandatory) a string providing the name of the cloud account. This name can be used in a builder section to reference the rest of the cloud account information.
- `queryId` (mandatory): a string providing your Eucalyptus query id. To get this key, login into your Eucalyptus admin console (for example https://myserver.domain.com:8443). Go to Query Interface Credentials > Show keys, the query id will be displayed.
- `secretKey` (mandatory): a string of your your Eucalyptus secret key. To get this key, login into your Eucalyptus admin console (for example https://myserver.domain.com:8443). Go to Query Interface Credentials > Show keys, the secret key will be displayed
- `x509PrivateKey` (mandatory): a string providing the location of the X.509 certificate private key. This may be a path or URL. This is the private key of the X.509 certificate. To get an X.509 private key, login into your

Eucalyptus admin console, go to Credentials ZIP-file and click on the button Download credentials. Unzip this file, you should find the private key with the name XXXX-XXXX-XXXX-pk.pem.

- `x509Cert` (mandatory): a string providing the location of the X.509 certificate public key. This may be a path or URL. To get a X.509 certificate, login into your Eucalyptus admin console, go to the Credentials ZIP-file and click on the button Download credentials. Unzip this file, you should find the certificate with the name XXXX-XXXX-XXXX-cert.pem

Note: In the case where `name` or `file` is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Example

The following example shows an Eucalyptus builder with all the information to build and publish a machine image to Eucalyptus.

If you are using YAML:

```
---
builders:
- type: Eucalyptus KVM
  account:
    type: Eucalyptus
    name: My Eucalyptus Account
    accountNumber: '111122223333'
    x509PrivateKey: "/home/joris/accounts/euca/euca-pk.pem"
    x509Cert: "/home/joris/accounts/euca/euca-cert.pem"
    cloudCert: "/home/joris/accounts/euca/cloud-cert.pem"
    endpoint: http://127.0.0.1/8773
    queryId: WkVpyXXZ77rXcdeSbds3lkXcr5Jc4GeUtkA
    secretKey: ir9CKRvOXXTHJXXj8VPRXX7PgxxY9DY0VLng
  imageName: CentOS Core
  description: CentOS Base Image
  bucket: ussprodbucket
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Eucalyptus KVM",
      "account": {
        "type": "Eucalyptus",
        "name": "My Eucalyptus Account",
        "accountNumber": "111122223333",
        "x509PrivateKey": "/home/joris/accounts/euca/euca-pk.pem",
        "x509Cert": "/home/joris/accounts/euca/euca-cert.pem",
        "cloudCert": "/home/joris/accounts/euca/cloud-cert.pem",
        "endpoint": "http://127.0.0.1/8773",
        "queryId": "WkVpyXXZ77rXcdeSbds3lkXcr5Jc4GeUtkA",
        "secretKey": "ir9CKRvOXXTHJXXj8VPRXX7PgxxY9DY0VLng"
      },
      "imageName": "CentOS Core",
      "description": "CentOS Base Image",
      "bucket": "ussprodbucket"
    }
```

```
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a YAML file `euca-account.yml`.

```
---
accounts:
- type: Eucalyptus
  name: My Eucalyptus Account
  accountNumber: '111122223333'
  x509PrivateKey: "/home/joris/accounts/euca/euca-pk.pem"
  x509Cert: "/home/joris/accounts/euca/euca-cert.pem"
  cloudCert: "/home/joris/accounts/euca/cloud-cert.pem"
  endpoint: http://127.0.0.1/8773
  queryId: WkVpyXXZ77rXcdeSbds3lkXcr5Jc4GeUtkA
  secretKey: ir9CKRvOXXTHJXXj8VPRXX7PgxxY9DY0VLng
```

If you are using JSON, create a JSON file `euca-account.json`:

```
{
  "accounts": [
    {
      "type": "Eucalyptus",
      "name": "My Eucalyptus Account",
      "accountNumber": "111122223333",
      "x509PrivateKey": "/home/joris/accounts/euca/euca-pk.pem",
      "x509Cert": "/home/joris/accounts/euca/euca-cert.pem",
      "cloudCert": "/home/joris/accounts/euca/cloud-cert.pem",
      "endpoint": "http://127.0.0.1/8773",
      "queryId": "WkVpyXXZ77rXcdeSbds3lkXcr5Jc4GeUtkA",
      "secretKey": "ir9CKRvOXXTHJXXj8VPRXX7PgxxY9DY0VLng"
    }
  ]
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: Eucalyptus KVM
  account:
    file: "/home/joris/accounts/euca-account.yml"
  imageName: CentOS Core
  description: CentOS Base Image
  bucket: ussprodbucket
```

If you are using JSON:

```
{
  "builders": [
```

```
    {
      "type": "Eucalyptus KVM",
      "account": {
        "file": "/home/joris/accounts/euca-account.json"
      },
      "imageName": "CentOS Core",
      "description": "CentOS Base Image",
      "bucket": "ussprodbucket"
    }
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: Eucalyptus KVM
  account:
    name: My Eucalytpus Account
  imageName: CentOS Core
  description: CentOS Base Image
  bucket: ussprodbucket
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Eucalyptus KVM",
      "account": {
        "name": "My Eucalytpus Account"
      },
      "imageName": "CentOS Core",
      "description": "CentOS Base Image",
      "bucket": "ussprodbucket"
    }
  ]
}
```

Flexiant

Default builder type: `Flexiant QCOW2 - KVM/Xen/VMware`, `Flexiant OVA - VMware` or `Flexiant RAW - KVM/Xen`

Require Cloud Account: Yes

flexiant.com

The Flexiant builder provides information for building and publishing the machine image to a Flexiant cloud platform. This builder supports KVM (`Flexiant QCOW2 - KVM/Xen/VMware`), VMware (`Flexiant OVA - VMware`) or Raw (`Flexiant RAW - KVM/Xen`) based images for Flexiant.

These builder types are the default names provided by UForge AppCenter.

Note: These builder type names can be changed by your UForge administrator. To get the available builder types,

please refer to *format*

The Flexiant builder requires cloud account information to upload and register the machine image to the Flexiant platform.

The Flexiant builder section has the following definition when using YAML:

```
---
builders:
- type: Flexiant RAW - KVM/Xen
  # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Flexiant OVA - VMware",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Flexiant: `Flexiant QCOW2 - KVM/Xen/VMware`, `Flexiant OVA - VMware` or `Flexiant RAW - KVM/Xen`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. If an OVF r**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
- **installation (optional): an object providing low-level installation or first boot options. These override any installation optio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Flexiant: `Flexiant QCOW2 - KVM/Xen/VMware`, `Flexiant OVA - VMware` or `Flexiant RAW - KVM/Xen`. To get the available builder type, please refer to *format*

- `account` (mandatory): an object providing the Flexiant cloud account information required to publish the built machine image.
- `virtualDatacenterName` (mandatory): a string providing the datacenter name where to register the machine image. Note, the user must have access to this datacenter.
- `machineImageName` (mandatory): a string providing the name of the machine image to displayed.
- `diskOffering` (mandatory): a string providing the disk offering to register the machine image under.

Flexiant Cloud Account

Key: `account` Used to authenticate the Flexiant platform.

The Flexiant cloud account has the following valid keys:

- `type` (mandatory): a string providing the cloud account type. Default platform type for Flexiant is `Flexiant`. To get the available platform type, please refer to *platform*
- `name` (mandatory): a string providing the name of the cloud account. This name can be used in a builder section to reference the rest of the cloud account information.
- `apiUsername` (mandatory): a string providing your API username. To get your api username, log in to Flexiant cloud orchestrator, click on Settings > Your API Details
- `password` (mandatory): a string providing your Flexiant cloud orchestrator account password
- `wsdlUrl` (mandatory): a string providing the wsdl URL of the Flexiant cloud orchestrator, for example: https://myapi.example2.com:4442/?wsdl
- `file` (optional): a string providing the location of the account information. This can be a pathname (relative or absolute) or an URL.

Note: In the case where `name` or `file` is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Example

The following example shows a Flexiant builder with all the information to build and publish a machine image to the Flexiant.

If you are using YAML:

```
---
builders:
- type: Flexiant RAW - KVM/Xen
  account:
    type: Flexiant
    name: My Flexiant account
    apiUsername: name@domain.com/mykey1111
    password: mypassword
    wsdlUrl: myWsdlurl
  hardwareSettings:
    memory: 1024
  installation:
    diskSize: 2000
  virtualDatacenterName: KVM (CEPH Cluster)
```

```
  machineImageName: test_hammr
  diskOffering: 21 GB
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Flexiant RAW - KVM/Xen",
      "account": {
        "type": "Flexiant",
        "name": "My Flexiant account",
        "apiUsername": "name@domain.com/mykey1111",
        "password": "mypassword",
        "wsdlUrl": "myWsdlurl"
      },
      "hardwareSettings": {
        "memory": 1024
      },
      "installation": {
        "diskSize": 2000
      },
      "virtualDatacenterName": "KVM (CEPH Cluster)",
      "machineImageName": "test_hammr",
      "diskOffering": "21 GB"
    }
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a YAML file `Flexiant-account.yml`.

```
---
accounts:
- type: Flexiant
  name: My Flexiant account
  apiUsername: name@domain.com/mykey1111
  password: mypassword
  wsdlUrl: myWsdlurl
```

If you are using JSON, create a JSON file `Flexiant-account.json`:

```
{
  "accounts": [
    {
      "type": "Flexiant",
      "name": "My Flexiant account",
      "apiUsername": "name@domain.com/mykey1111",
      "password": "mypassword",
      "wsdlUrl": "myWsdlurl"
    }
  ]
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: Flexiant RAW - KVM/Xen
  account:
    file: "/path/to/flexiant-account.yml"
  hardwareSettings:
    memory: 1024
  installation:
    diskSize: 2000
  virtualDatacenterName: KVM (CEPH Cluster)
  machineImageName: test_hammr
  diskOffering: 21 GB
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Flexiant RAW - KVM/Xen",
      "account": {
        "file": "/path/to/flexiant-account.json"
          },
      "hardwareSettings": {
        "memory": 1024
      },
      "installation": {
        "diskSize": 2000
      },
      "virtualDatacenterName": "KVM (CEPH Cluster)",
      "machineImageName": "test_hammr",
      "diskOffering": "21 GB"
    }
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: Flexiant RAW - KVM/Xen
  account:
    name: My Flexiant Account
  hardwareSettings:
    memory: 1024
  installation:
    diskSize: 2000
  virtualDatacenterName: KVM (CEPH Cluster)
  machineImageName: test_hammr
  diskOffering: 21 GB
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Flexiant RAW - KVM/Xen",
      "account": {
        "name": "My Flexiant Account"
          },
      "hardwareSettings": {
        "memory": 1024
      },
      "installation": {
        "diskSize": 2000
      },
      "virtualDatacenterName": "KVM (CEPH Cluster)",
      "machineImageName": "test_hammr",
      "diskOffering": "21 GB"
    }
  ]
}
```

Google Compute Engine

Default builder type: `Google Compute Engine`

Require Cloud Account: Yes

Google Compute Engine

The GCE builder provides information for building and publishing the machine image for Googe Compute Engine. The GCE builder requires cloud account information to upload and register the machine image to Google Compute Engine public cloud. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The GCE builder section has the following definition when using YAML:

```
---
builders:
- type: Google Compute Engine
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Google Compute Engine",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Google compute Engine: `Google Compute Engine`. To get the available builder type, please refer to *format*
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Google compute Engine: `Google Compute Engine`. To get the available builder type, please refer to *format*
- `account` (mandatory): an object providing the GCE cloud account information required to publish the built machine image.
- `bucket` (mandatory): a string providing the bucket name where to store the machine image. The bucket name can only contain lower case alpha characters [a-z] and the special character "-".
- `bucketLocation` (mandatory): a string providing the bucket location where to store the machine image. See below for valid values.
- `computeZone` (mandatory): a string providing the compute zone where this machine image will be used. See below for valid compute zone values.
- `description` (optional): a string providing the description for the machine image.
- `diskNamePrefix` (mandatory): a string providing the disk name prefix used when creating the disks for the running machine (note the prefix name can only contain lower case alpha characters [a-z] and the special character "-")
- `projectId` (mandatory): a string providing the project Id to associate this machine image with.
- `storageClass` (mandatory): a string providing the storage type to use with this machine image. See below for valid storage class values

Valid Compute Zones

The following zones are supported:

- `us-central1-a`: US (availability zone: a)
- `us-central1-b`: US (availability zone: b)
- `europe-west1-a`: Europe (availability zone: a)
- `europe-west1-a`: Europe (availability zone: b)

Valid Bucket Locations

The following bucket locations are supported:

- `EU`
- `US`

Valid Storage Classes

The following storage classes are supported:

- `STANDARD`
- `DURABLE_REDUCED_AVAILABILITY`

GCE Cloud Account

Key: `account` Used to authenticate to GCE.

The GCE cloud account has the following valid keys:

- `type` (mandatory): a string providing the cloud account type. Default platform type for Google Compute Engine: `Google Compute Engine`. To get the available platform type, please refer to *platform*
- `certPassword` (mandatory): A string providing the password to decrypt the GCE certificate. This password is normally provided along with the certificate.
- `cert` (mandatory): A string providing the pathname or URL where to retrieve your GCE certificate. This should be a (.pem) file.
- `name` (mandatory): a string providing the name of the cloud account. This name can be used in a builder section to reference the rest of the cloud account information.

Note: In the case where `name` or `file` is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Example

The following example shows a GCE builder with all the information to build and publish a machine image to Google Compute Engine.

If you are using YAML:

```
---
builders:
- type: Google Compute Engine
  account:
    type: Google Compute Engine
    name: My GCE Account
    username: joris
    certPassword: myCertPassword
    cert: "/home/joris/certs/gce.pem"
  computeZone: europe-west1-a
  bucketLocation: EU
```

```
  bucket: jorisbucketname
  projectId: jorisproject
  storageClass: STANDARD
  diskNamePrefix: uss-
  description: CentOS Core machine image
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Google Compute Engine",
      "account": {
        "type": "Google Compute Engine",
        "name": "My GCE Account",
        "username": "joris",
        "certPassword": "myCertPassword",
        "cert": "/home/joris/certs/gce.pem"
      },
      "computeZone": "europe-west1-a",
      "bucketLocation": "EU",
      "bucket": "jorisbucketname",
      "projectId": "jorisproject",
      "storageClass": "STANDARD",
      "diskNamePrefix": "uss-",
      "description": "CentOS Core machine image"
    }
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a YAML file `gce-account.yml`.

```
---
accounts:
- type: Google Compute Engine
  name: My GCE Account
  username: joris
  certPassword: myCertPassword
  cert: "/home/joris/certs/gce.pem"
```

If you are using JSON, create a JSON file `gce-account.json`:

```
{
  "accounts": [
    {
        "type": "Google Compute Engine",
        "name": "My GCE Account",
        "username": "joris",
        "certPassword": "myCertPassword",
        "cert": "/home/joris/certs/gce.pem"
    }
  ]
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: Google Compute Engine
  account:
    file: "/home/joris/accounts/gce-account.yml"
  computeZone: europe-west1-a
  bucketLocation: EU
  bucket: jorisbucketname
  projectId: jorisproject
  storageClass: STANDARD
  diskNamePrefix: uss-
  description: CentOS Core machine image
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Google Compute Engine",
      "account": {
        "file": "/home/joris/accounts/gce-account.json"
      },
      "computeZone": "europe-west1-a",
      "bucketLocation": "EU",
      "bucket": "jorisbucketname",
      "projectId": "jorisproject",
      "storageClass": "STANDARD",
      "diskNamePrefix": "uss-",
      "description": "CentOS Core machine image"
    }
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: Google Compute Engine
  account:
    name: My GCE Account
  computeZone: europe-west1-a
  bucketLocation: EU
  bucket: jorisbucketname
  projectId: jorisproject
  storageClass: STANDARD
  diskNamePrefix: uss-
  description: CentOS Core machine image
```

If you are using JSON:

```
{
  "builders": [
```

```
    {
      "type": "Google Compute Engine",
      "account": {
        "name": "My GCE Account"
      },
      "computeZone": "europe-west1-a",
      "bucketLocation": "EU",
      "bucket": "jorisbucketname",
      "projectId": "jorisproject",
      "storageClass": "STANDARD",
      "diskNamePrefix": "uss-",
      "description": "CentOS Core machine image"
    }
  ]
}
```

Microsoft Azure

Default builder type: `Microsoft Azure`

Require Cloud Account: Yes

azure.microsoft.com

The Azure builder provides information for building and publishing the machine image to the Microsoft Azure cloud platform. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The Azure builder section has the following definition when using YAML:

```
---
builders:
- type: Microsoft Azure
  # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Microsoft Azure",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Azure: `Microsoft Azure`. To get the available builder type, please refer to *format*

- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**

  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the *Stack*. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Azure: `Microsoft Azure`. To get the available builder type, please refer to *format*
- `account` (mandatory): an object providing all the cloud account information to authenticate and publish a machine image to Azure.
- `region` (mandatory): a string providing the region where to create the storage account. If the storage account already exists, then you should not specify a region. See below for valid regions.
- `storageAccount` (mandatory): a string providing the storage account to use for uploading and storing the machine image. The storage account is the highest level of the namespace for accessing each of the fundamental services.

Valid Azure Regions

- `North Central US`
- `South Central US`
- `East US`
- `West US`
- `North Europe`
- `West Europe`
- `East Asia`

Azure Cloud Account

Key: `account`

Used to authenticate the Azure platform. The Azure cloud account has the following valid keys:

- `type` (mandatory): a string providing the cloud account type. Default platform type for Microsoft Azure: `Microsoft Azure`. To get the available platform type, please refer to *platform*
- `name` (mandatory): a string providing the name of the cloud account. This name can be used in a builder section to reference the rest of the cloud account information.

- `publishsettings` (mandatory): A string providing the pathname where to retrieve the publish settings and subscription information file. This should be a (.publishsettings) file.
- `file` (optional): a string providing the location of the account information. This can be a pathname (relative or absolute) or an URL.

Note: In the case where name or file is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Example

The following example shows an Azure builder with all the information to build and publish a machine image to Azure.

If you are using YAML:

```
---
builders:
- type: Microsoft Azure
  account:
    type: Microsoft Azure
    name: My Azure account
    publishsettings: "/path/to/Pay-As-You-Go-4-25-2016-credentials.publishsettings"
  storageAccount: mystorageaccount
  region: Central US
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Microsoft Azure",
      "account": {
        "type": "Microsoft Azure",
        "name": "My Azure account",
        "publishsettings": "/path/to/Pay-As-You-Go-4-25-2016-credentials.
↪publishsettings"
      },
      "storageAccount":"mystorageaccount",
      "region":"Central US"
    }
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a YAML file `azure-account.yml`.

```
---
accounts:
- type: Microsoft Azure
```

```
  name: My Azure account
  publishsettings: "/path/to/Pay-As-You-Go-date-credentials.publishsettings"
```

If you are using JSON, create a JSON file `azure-account.json`:

```
{
  "accounts": [
    {
      "type": "Microsoft Azure",
      "name": "My Azure account",
      "publishsettings": "/path/to/Pay-As-You-Go-date-credentials.publishsettings"
    }
  ]
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: Microsoft Azure
  account:
    file: "/home/joris/accounts/azure-account.yml"
  storageAccount: mystorageaccount
  region: Central US
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Microsoft Azure",
      "account": {
            "file": "/home/joris/accounts/azure-account.json"
      },
      "storageAccount":"mystorageaccount",
      "region":"Central US"
    }
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: Microsoft Azure
  account:
    name: My Azure Account
  storageAccount: mystorageaccount
  region: Central US
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Microsoft Azure",
      "account": {
            "name": "My Azure Account"
      },
      "storageAccount":"mystorageaccount",
      "region":"Central US"
    }
  ]
}
```

Fujitsu K5

Default builder type: `Fujitsu K5`

Require Cloud Account: Yes

www.fujitsu.com/global/solutions/cloud/k5

The K5 builder provides information for building and publishing the machine image to the Fujitsu K5 cloud platform. This builder supports only VMDK (`Fujitsu K5`) based images for K5. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder type, please refer to *format*.

The K5 builder requires cloud account information to upload and register the machine image to the K5 platform.

The K5 builder section has the following definition when using YAML:

```
---
builders:
- type: Fujitsu K5
  # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Fujitsu K5",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for K5: `Fujitsu K5`. To get the available builder type, please refer to *format*.

- `hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. The following valid keys for hardware settings are: * `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB). * `hwType` (optional): an integer providing the hardware type for the machine image. This is the VMware hardware type: 4 (ESXi>3.x), 7 (ESXi>4.x) or 9 (ESXi>5.x)
- `installation` (optional): an object providing low-level installation or first boot options. These override any installation options in the *Stack* section. The following valid keys for installation are: * `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for K5: `Fujitsu K5`. To get the available builder type, please refer to *format*.
- `account` (mandatory): an object providing the K5 cloud account information required to publish the built machine image.
- `displayName` (mandatory): a string providing the name of the image that will be displayed.
- `domain` (mandatory): a string providing the K5 domain to publish this machine image to.
- `project` (mandatory): a string providing the K5 project to publish this machine image to.
- `region` (mandatory): a string providing the region where to publish the machine image. See below for valid regions.

Valid Regions

The following regions are supported:

- `uk-1`: United Kingdom Region 1
- `jp-east-1`: Eastern Japan Region 1
- `jp-west-1`: Western Japan Region 1
- `jp-west-2`: Western Japan Region 2

K5 Cloud Account

Key: `account`

Used to authenticate the K5 platform.

The K5 cloud account has the following valid keys:

- `type` (mandatory): a string providing the cloud account type. Default platform type for K5 is `K5`. To get the available platform type, please refer to *platform*

- `name` (mandatory): a string providing the name of the cloud account. This name can be used in a builder section to reference the rest of the cloud account information.
- `login` (mandatory): a string providing the user for authenticating to keystone for publishing images
- `password` (mandatory): a string providing the password for authenticating to keystone for publishing images
- `file` (optional): a string providing the location of the account information. This can be a pathname (relative or absolute) or an URL.

Note: In the case where `name` or `file` is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Example

The following example shows a K5 builder with all the information to build and publish a machine image to K5.

If you are using YAML:

```
---
builders:
- type: Fujitsu K5
  account:
    type: K5
    name: My K5 Account
    login: mylogin
    password: mypassword
  displayName: K5_testHammr
  domain: mydomain
  project: myproject
  region: uk-1
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Fujitsu K5",
      "account": {
        "type": "K5",
        "name": "My K5 Account",
        "login": "mylogin",
        "password": "mypassword"
      },
      "displayName": "K5_testHammr",
      "domain": "mydomain",
      "project": "myproject",
      "region": "uk-1"
    }
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a YAML file `k5-account.yml`.

```
---
accounts:
- type: K5
  name: My K5 Account
  login: mylogin
  password: mypassword
```

If you are using JSON, create a JSON file `k5-account.json`:

```
{
  "accounts": [
    {
      "type": "K5",
      "name": "My K5 Account",
      "login": "mylogin",
      "password": "mypassword"
    }
  ]
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: Fujitsu K5
  account:
    file: "/path/to/k5-account.yml"
  displayName: K5_testHammr
  domain: mydomain
  project: myproject
  region: uk-1
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Fujitsu K5",
      "account": {
            "file": "/path/to/k5-account.json"
      },
      "displayName": "K5_testHammr",
      "domain": "mydomain",
      "project": "myproject",
      "region": "uk-1"
    }
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: Fujitsu K5
  account:
    name: My K5 Account
  displayName: K5_testHammr
  domain: mydomain
  project: myproject
  region: uk-1
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Fujitsu K5",
      "account": {
        "name": "My K5 Account"
        },
      "displayName": "K5_testHammr",
      "domain": "mydomain",
      "project": "myproject",
      "region": "uk-1"
    }
  ]
}
```

Nimbula

Default builder type: `Nimbula ESX` or `Nimbula KVM`

Require Cloud Account: Yes

The Nimbula builder provides information for building and publishing the machine image to the Nimbula cloud platform. This builder supports KVM (`Nimbula KVM`) or VMware (`Nimbula ESX`) based images for Nimbula. These builder types are the default names provided by UForge AppCenter.

Note: These builder type names can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The Nimbula builder requires cloud account information to upload and register the machine image to the Nimbula platform. The Nimbula builder section has the following definition when using YAML:

```
---
builders:
- type: Nimbula KVM
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Nimbula KVM",
```

```
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Nimbula: `Nimbula ESX` or, ``Nimbula KVMK`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. If an OVF r**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Nimbula: `Nimbula ESX` or, ``Nimbula KVMK`. To get the available builder type, please refer to *format*
- `account` (mandatory): an object providing the Nimbula cloud account information required to publish the built machine image.
- `description` (mandatory): a string providing the description that will be displayed for the machine image.
- `imageListName` (mandatory): a string providing the list name where to register the machine image. Note that this is the full pathname, for example `/usharesoft/administrator/myimages`. Machine images can be added to an image list to create a versioned selection of related machine images recording the versions of the image over its lifetime. An image maintainer can add newer versions of a machine image to the image list and can set the default version to be used when this image list is invoked in a launch plan to deploy VM instances
- `imageVersion` (mandatory): a string providing the version of the machine image being registered.

Nimbula Cloud Account

Key: `account`

Used to authenticate the Nimbula platform. The Nimbula cloud account has the following valid keys:

- `type` (mandatory): a string providing the cloud account type. Default platform type for Nimbula is `Nimbula`. To get the available platform type, please refer to *platform*
- `file` (optional): a string providing the location of the account information. This can be a pathname (relative or absolute) or an URL.
- `endpoint` (mandatory): URL endpoint of the Nimbula cloud
- `name` (mandatory): a string providing the name of the cloud account. This name can be used in a builder section to reference the rest of the cloud account information.
- `password` (mandatory): a string providing the password used to to authenticate to Nimbula Director
- `username` (mandatory): a string providing the user used to authenticate to Nimbula Director. This is in the form of a URI, for example `/root/root`

Note: In the case where `name` or `file` is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Example

The following example shows an Nimbula builder with all the information to build and publish a machine image to Nimbula.

If you are using YAML:

```
---
builders:
- type: Nimbula KVM
  account:
    type: Nimbula
    name: My Nimbula Account
    endpoint: http://20.20.20.201
    username: myLogin
    password: myPassWD
  hardwareSettings:
    memory: 1024
  installation:
    diskSize: 2000
  imageListName: "/usharesoft/administrator/myimages"
  imageVersion: '1'
  description: CentOS Core Image
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Nimbula KVM",
      "account": {
        "type": "Nimbula",
        "name": "My Nimbula Account",
        "endpoint": "http://20.20.20.201",
        "username": "myLogin",
        "password": "myPassWD"
```

```
      },
      "hardwareSettings": {
        "memory": 1024
      },
      "installation": {
        "diskSize": 2000
      },
      "imageListName": "/usharesoft/administrator/myimages",
      "imageVersion": "1",
      "description": "CentOS Core Image"
    }
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a YAML file `nimbula-account.yml`.

```
---
accounts:
- type: Nimbula
  name: My Nimbula Account
  endpoint: http://20.20.20.201
  username: myLogin
  password: myPassWD
```

If you are using JSON, create a JSON file `nimbula-account.json`:

```
{
  "accounts": [
    {
        "type": "Nimbula",
        "name": "My Nimbula Account",
        "endpoint": "http://20.20.20.201",
        "username": "myLogin",
        "password": "myPassWD"
    }
  ]
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: Nimbula KVM
  account:
    file: "/home/joris/accounts/nimbula-account.yml"
  hardwareSettings:
    memory: 1024
  installation:
    diskSize: 2000
```

```
  imageListName: "/usharesoft/administrator/myimages"
  imageVersion: '1'
  description: CentOS Core Image
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Nimbula KVM",
      "account": {
        "file": "/home/joris/accounts/nimbula-account.json"
      },
      "hardwareSettings": {
        "memory": 1024
      },
      "installation": {
        "diskSize": 2000
      },
      "imageListName": "/usharesoft/administrator/myimages",
      "imageVersion": "1",
      "description": "CentOS Core Image"
    }
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: Nimbula KVM
  account:
    name: My Nimbula Account
  hardwareSettings:
    memory: 1024
  installation:
    diskSize: 2000
  imageListName: "/usharesoft/administrator/myimages"
  imageVersion: '1'
  description: CentOS Core Image
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Nimbula KVM",
      "account": {
        "name": "My Nimbula Account"
      },
      "hardwareSettings": {
        "memory": 1024
      },
      "installation": {
        "diskSize": 2000
      },
```

```
      "imageListName": "/usharesoft/administrator/myimages",
      "imageVersion": "1",
      "description": "CentOS Core Image"
    }
  ]
}
```

OpenStack

Default builder type: `OpenStack QCOW2`, `OpenStack VMDK`, `OpenStack VHD` or `OpenStack VDI`

Require Cloud Account: Yes

www.openstack.org

The OpenStack builder provides information for building and publishing the machine image to the OpenStack cloud platform. This builder supports KVM (`OpenStack QCOW2`), VMware (`OpenStack VMDK`), Microsoft (`OpenStack VHD`) or VirtualBox (`OpenStack VDI`) based images for Openstack. These builder types are the default names provided by UForge AppCenter.

Note: These builder type names can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The OpenStack builder requires cloud account information to upload and register the machine image to the OpenStack platform.

The OpenStack builder section has the following definition when using YAML:

```
---
builders:
- type: OpenStack QCOW2
  # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "OpenStack QCOW2",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for OpenStack: `OpenStack QCOW2`, `OpenStack VMDK`, `OpenStack VDI` or `OpenStack VHD`. To get the available builder type, please refer to *format*
- `installation` (optional): an object providing low-level installation or first boot options. These override any installation options in the *Stack* section. The following valid keys for installation are: * `diskSize` (manda-

tory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for OpenStack: `OpenStack QCOW2`, `OpenStack VMDK`, `OpenStack VDI` or `OpenStack VHD`. To get the available builder type, please refer to *format*
- `account` (mandatory): an object providing the OpenStack cloud account information required to publish the built machine image.
- `displayName` (mandatory): a string providing the name of the image that will be displayed.
- `tenantName` (mandatory for keystone v2.0): a string providing the name of the tenant to register the machine image to. This value is ony required if the cloud account's `keystoneVersion` is `v2.0`
- `keystoneDomain` (mandatory for keystone v3.0): a string providing the keystone domain to publish this machine image to.
- `keystoneProject` (mandatory for keystone v3.0): a string providing the keystone project to publish this machine image to.
- `publicImage` (optional): a boolean to determine if the machine image is public (for other users to use for provisioning).

OpenStack Cloud Account

Key: `account`

Used to authenticate the OpenStack platform.

The OpenStack cloud account has the following valid keys:

- `type` (mandatory): a string providing the cloud account type. Default platform type for Openstack is `OpenStack`. To get the available platform type, please refer to *platform*
- `name` (mandatory): a string providing the name of the cloud account. This name can be used in a builder section to reference the rest of the cloud account information.
- `glanceUrl` (mandatory): a string providing the API URL endpoint of the OpenStack glance service. For example: http://www.example.com/v1/
- `keystoneUrl` (mandatory): a string providing the URL endpoint for the OpenStack keystone service to authenticate with. For example: http://www.example.com:5000
- `keystoneVersion` (mandatory): a string providing the keystone version of the OpenStack platform. Refer to *Valid Keystone Versions* for the valid keystone versions.
- `login` (mandatory): a string providing the user for authenticating to keystone for publishing images
- `password` (mandatory): a string providing the password for authenticating to keystone for publishing images

- `file` (optional): a string providing the location of the account information. This can be a pathname (relative or absolute) or an URL.

Note: In the case where `name` or `file` is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Valid Keystone Versions

- `v2.0`: Keystone version 2.0
- `3.0` : Keystone version 3.0

Example

The following example shows an OpenStack builder with all the information to build and publish a machine image to OpenStack.

If you are using YAML:

```
---
builders:
- type: OpenStack QCOW2
  account:
    type: OpenStack
    name: My OpenStack Account
    glanceUrl: http://myglanceurl/v1/
    keystoneUrl: http://mykeystoneurl:9292/v1
    keystoneVersion: v2.0
    login: mylogin
    password: mypassword
  displayName: OpenStack_testHammr
  tenantName: mytenant
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "OpenStack QCOW2",
      "account": {
        "type": "OpenStack",
        "name": "My OpenStack Account",
        "glanceUrl": "http://myglanceurl/v1/",
        "keystoneUrl": "http://mykeystoneurl:9292/v1",
        "keystoneVersion": "v2.0",
        "login": "mylogin",
        "password": "mypassword"
      },
      "displayName": "OpenStack_testHammr",
      "tenantName": "mytenant"
    }
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a YAML file `openstack-account.yml`.

```
---
accounts:
- type: OpenStack
  name: My OpenStack Account
  glanceUrl: http://myglanceurl/v1/
  keystoneUrl: http://mykeystoneurl:9292/v1
  keystoneVersion: http://mykeystoneversion:5000/v2.0
  login: mylogin
  password: mypassword
```

If you are using JSON, create a JSON file `openstack-account.json`:

```
{
  "accounts": [
    {
      "type": "OpenStack",
      "name": "My OpenStack Account",
      "glanceUrl": "http://myglanceurl/v1/",
      "keystoneUrl": "http://mykeystoneurl:9292/v1",
      "keystoneVersion": "http://mykeystoneversion:5000/v2.0",
      "login": "mylogin",
      "password": "mypassword"
    }
  ]
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: OpenStack QCOW2
  account:
    file: "/path/to/openstack-account.yml"
  displayName: OpenStack_testHammr
  tenantName: mytenant
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "OpenStack QCOW2",
      "account": {
            "file": "/path/to/openstack-account.json"
      },
      "displayName": "OpenStack_testHammr",
      "tenantName": "mytenant"
    }
```

```
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: OpenStack QCOW2
  account:
    name: My OpenStack Account
  displayName: OpenStack_testHammr
  tenantName: mytenant
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "OpenStack QCOW2",
      "account": {
        "name": "My OpenStack Account"
        },
      "displayName": "OpenStack_testHammr",
      "tenantName": "mytenant"
    }
  ]
}
```

Outscale

Default builder type: `Outscale`

Require Cloud Account: Yes

outscale.com

The Outscale builder provides information for building and publishing the machine image for Outscale cloud platform. The Outscale builder requires cloud account information to upload and register the machine image to Outscale cloud. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The Outscale builder section has the following definition when using YAML:

```
---
builders:
- type: Outscale
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
```

```
    {
      "type": "Outscale",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Outscale: `Outscale`. To get the available builder type, please refer to *format*
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. If the machine image is to be stored in Amazon S3, the maximum disk size is 10GB, otherwise if this is an EBS-backed machine image the maximum disk size is 1TB.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Outscale: `Outscale`. To get the available builder type, please refer to *format*
- `account` (mandatory): an object providing the AWS cloud account information required to publish the built machine image.
- `zone` (mandatory): a string providing the region where to publish the machine image. See below for valid regions.

Valid Regions

The following regions are supported:

- `ap-northeast-1`: Asia Pacific (Tokyo) Region
- `ap-southeast-1`: Asia Pacific (Singapore) Region
- `eu-west-1`: EU (Ireland) Region
- `sa-east-1`: South America (Sao Paulo) Region
- `us-east-1`: US East (North Virginia) Region
- `us-west-1`: US West (North california) Region

- `us-west-2`: US West (Oregon) Region

Outscale Cloud Account

Key: `account` Used to authenticate to Outscale cloud platform.

The Outscale cloud account has the following valid keys:

- `type` (mandatory): a string providing the cloud account type. Default platform type for Outscale: `Outscale`. To get the available platform type, please refer to *platform*
- `name` (mandatory): a string providing the name of the cloud account. This name can be used in a `builder` section to reference the rest of the cloud account information.
- `secretAccessKey` (mandatory): A string providing your Outscale secret access key
- `accessKey` (mandatory): A string providing your Outscale access key id

---

Note: In the case where `name` or `file` is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Example

The following example shows an amazon builder with all the information to build and publish a machine image to Amazon EC2.

If you are using YAML:

```
---
builders:
- type: Outscale
  account:
    type: Outscale
    name: My Outscale Account
    accessKey: 789456123ajdiewjd
    secretAccessKey: ks30hPeH1xWqilJ04
  installation:
    diskSize: 10240
  zone: eu-west-2
  description: centos-template
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Outscale",
      "account": {
        "type": "Outscale",
        "name": "My Outscale Account",
        "accessKey": "789456123ajdiewjd",
        "secretAccessKey": "ks30hPeH1xWqilJ04"
      },
      "installation": {
        "diskSize": 10240
      },
```

```
      "zone": "eu-west-2",
      "description": "centos-template"
    }
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a YAML file `outscale-account.yml`.

```
---
accounts:
- type: Outscale
  name: My Outscale Account
  accessKey: 789456123ajdiewjd
  secretAccessKey: ks30hPeH1xWqilJ04
```

If you are using JSON, create a JSON file `outscale-account.json`:

```
{
  "accounts": [
    {
      "type": "Outscale",
      "name": "My Outscale Account",
      "accessKey": "789456123ajdiewjd",
      "secretAccessKey": "ks30hPeH1xWqilJ04"
    }
  ]
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: Outscale
  account:
    file: "/home/joris/accounts/outscale-account.yml"
  installation:
    diskSize: 10240
  region: eu-west-2
  s3bucket: centos-template
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Outscale",
      "account": {
        "file": "/home/joris/accounts/outscale-account.json"
      },
```

```
      "installation": {
        "diskSize": 10240
      },
      "region": "eu-west-2",
      "s3bucket": "centos-template"
    }
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: Outscale
  account:
    name: My Outscale Account
  installation:
    diskSize: 10240
  region: eu-west-2
  s3bucket: centos-template
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Outscale",
      "account": {
        "name": "My Outscale Account"
      },
      "installation": {
        "diskSize": 10240
      },
      "region": "eu-west-2",
      "s3bucket": "centos-template"
    }
  ]
}
```

Suse Cloud

Default builder type: `SUSE Cloud`

Require Cloud Account: Yes

SuseCloud

The SuseCloud builder provides information for building and publishing the machine image to the SuseCloud cloud platform. The SuseCloud builder requires cloud account information to upload and register the machine image to the SuseCloud platform. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The SuseCloud builder section has the following definition when using YAML:

```
---
builders:
- type: Suse Cloud
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "SUSE Cloud",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Suse Cloud: `SUSE Cloud`. To get the available builder type, please refer to *format*
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Suse Cloud: `SUSE Cloud`. To get the available builder type, please refer to *format*
- `account` (mandatory): an object providing the SuseCloud cloud account information required to publish the built machine image.
- `description` (optional): an object providing the description of the machine image.
- `imageName` (mandatory): a string providing the name of the image that will be displayed.
- `keystoneDomain` (mandatory for keystone v3.0): a string providing the keystone domain to publish this machine image to.
- `keystoneProject` (mandatory for keystone v3.0): a string providing the keystone project to publish this machine image to.

- `paraVirtMode` (optional) a boolean to determine if the machine should be provisioned in para-virtualised mode. By default, machine images are provisioned in full-virtualised mode
- `publicImage` (optional): a boolean to determine if the machine image is public (for other users to use for provisioning).
- `tenant` (mandatory for keystone v2.0): a string providing the name of the tenant to register the machine image to. This value is ony required if the cloud account's `keystoneVersion` is `v2.0`

SuseCloud Cloud Account

Key: `account` Used to authenticate the SuseCloud platform.

The SuseCloud cloud account has the following valid keys:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Suse Cloud, `Suse Cloud`. To get the available builder type, please refer to *format*
- `file` (optional): a string providing the location of the account information. This can be a pathname (relative or absolute) or an URL.
- `endpoint` (mandatory): a string providing the API URL endpoint of the SuseCloud glance service. For example: http://www.example.com:9292
- `keystoneEndpoint` (mandatory): a string providing the URL endpoint for the SuseCloud keystone service to authenticate with. For example: http://www.example.com:5000
- `keystoneVersion` (mandatory): a string providing the keystone version of the SuseCloud platform. Refer to *Valid Keystone Versions* for the valid keystone versions.
- `name` (mandatory): a string providing the name of the cloud account. This name can be used in a builder section to reference the rest of the cloud account information.
- `password` (mandatory): a string providing the password for authenticating to keystone for publishing images
- `username` (mandatory): a string providing the user for authenticating to keystone for publishing images

---

Note: In the case where `name` or `file` is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Valid Keystone Versions

- `v2.0`: Keystone version 2.0
- `3.0` : Keystone version 3.0

Example

The following example shows a SuseCloud builder with all the information to build and publish a machine image to SuseCloud.

If you are using YAML:

```
---
builders:
- type: Suse Cloud
```

```
  account:
    type: Suse Cloud
    name: My SuseCloud Account
    endpoint: http://ow2-04.xsalto.net:9292/v1
    keystoneEndpoint: http://ow2-04.xsalto.net:5000/v2.0
    username: test
    password: password
  tenant: opencloudware
  imageName: joris-test
  description: CentOS Core template.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Suse Cloud",
      "account": {
        "type": "Suse Cloud",
        "name": "My SuseCloud Account",
        "endpoint": "http://ow2-04.xsalto.net:9292/v1",
        "keystoneEndpoint": "http://ow2-04.xsalto.net:5000/v2.0",
        "username": "test",
        "password": "password"
      },
      "tenant": "opencloudware",
      "imageName": "joris-test",
      "description": "CentOS Core template."
    }
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a YAML file `susecloud-account.yml`.

```
---
accounts:
- type: Suse Cloud
  name: My SuseCloud Account
  endpoint: http://ow2-04.xsalto.net:9292/v1
  keystoneEndpoint: http://ow2-04.xsalto.net:5000/v2.0
  username: test
  password: password
```

If you are using JSON, create a JSON file `susecloud-account.json`:

```
{
  "accounts": [
    {
        "type": "Suse Cloud",
        "name": "My SuseCloud Account",
        "endpoint": "http://ow2-04.xsalto.net:9292/v1",
        "keystoneEndpoint": "http://ow2-04.xsalto.net:5000/v2.0",
```

```
        "username": "test",
        "password": "password"
    }
  ]
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: Suse Cloud
  account:
    file: "/home/joris/accounts/susecloud-account.yml"
  tenant: opencloudware
  imageName: joris-test
  description: CentOS Core template.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Suse Cloud",
      "account": {
        "file": "/home/joris/accounts/susecloud-account.json"
      },
      "tenant": "opencloudware",
      "imageName": "joris-test",
      "description": "CentOS Core template."
    }
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: Suse Cloud
  account:
    name: My SuseCloud Account
  tenant: opencloudware
  imageName: joris-test
  description: CentOS Core template.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Suse Cloud",
      "account": {
        "name": "My SuseCloud Account"
      },
```

```
      "tenant": "opencloudware",
      "imageName": "joris-test",
      "description": "CentOS Core template."
    }
  ]
}
```

VMware vCloud Director

Default builder type: `VMware vCloud Director`

Require Cloud Account: Yes

The VMware vCloud Director builder provides information for building VMware vCloud Director compatible machine images. The VMware VCD builder section has the following definition when using YAML:

```
---
builders:
- type: VMware vCloud Director
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "VMware vCloud Director",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for VMware vCloud Director: `VMware vCloud Director`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. The followi**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
  - `hwType` (optional): an integer providing the hardware type for the machine image. This is the VMware hardware type: 4 (ESXi>3.x), 7 (ESXi>4.x) or 9 (ESXi>5.x)
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for VMware vCloud Director: `VMware vCloud Director`. To get the available builder type, please refer to *format*
- `account` (mandatory): an object providing the vCloud Director cloud account information required to publish the built machine image.
- `catalogName` (mandatory): a string providing the name of the catalog to register the machine image. Catalogs contain references to virtual systems and media images.
- `imageName` (mandatory): a string providing the name of the machine image to display in VCD.
- `orgName` (mandatory): a string providing the name of the vCloud organization to register the machine image.

vCloud Director Cloud Account

Key: `account` Used to authenticate to VMware vCloud Director.

The VCD cloud account has the following valid keys:

- `type` (mandatory): a string providing the cloud account type. Default platform type for VMware vCloud Director: `VMware vCloud Director`. To get the available platform type, please refer to *platform*
- `hostname` (mandatory): a string providing the fully-qualified hostname or IP address of the vCloud Directory platform.
- `password` (mandatory): a string providing the password to use to authenticate to the vCloud Director platform
- `port` (optional): an integer providing the vCloud Director platform port number (by default: 443 is used).
- `proxyHostname` (optional): a string providing the fully qualified hostname or IP address of the proxy to reach the vCloud Director platform.
- `proxyPort` (optional): an integer providing the proxy port number to use to reach the vCloud Director platform.
- `username` (mandatory): a string providing the user name to use to authenticate to the vCloud Director platform

Note: In the case where `name` or `file` is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Example

The following example shows a VCD builder with all the information to build and publish a machine image to VMware vCloud Director.

If you are using YAML:

```
---
builders:
- type: VMware vCloud Director
  account:
    type: VMware vCloud Director
    name: My VCD Account
    hostname: 10.1.1.2
    username: joris
    password: mypassword
  hardwareSettings:
    memory: 1024
    hwType: 7
  installation:
    diskSize: 10240
  orgName: HQProd
  catalogName: myCatalog
  imageName: CentOS Core
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "VMware vCloud Director",
      "account": {
        "type": "VMware vCloud Director",
        "name": "My VCD Account",
        "hostname": "10.1.1.2",
        "username": "joris",
        "password": "mypassword"
      },
      "hardwareSettings": {
        "memory": 1024,
        "hwType": 7
      },
      "installation": {
        "diskSize": 10240
      },
      "orgName": "HQProd",
      "catalogName": "myCatalog",
      "imageName": "CentOS Core"
    }
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a yaml file `vcd-account.yml`.

```
---
accounts:
- type: VMware vCloud Director
  name: My VCD Account
  hostname: 10.1.1.2
  username: joris
```

```
  password: mypassword
```

If you are using JSON, create a JSON file `vcd-account.json`:

```
{
  "accounts": [
    {
      "type": "VMware vCloud Director",
      "name": "My VCD Account",
      "hostname": "10.1.1.2",
      "username": "joris",
      "password": "mypassword"
    }
  ]
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: VMware vCloud Director
  account:
    file: "/home/joris/accounts/vcd-account.yml"
  hardwareSettings:
    memory: 1024
    hwType: 7
  installation:
    diskSize: 10240
  orgName: HQProd
  catalogName: myCatalog
  imageName: CentOS Core
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "VMware vCloud Director",
      "account": {
        "file": "/home/joris/accounts/vcd-account.json"
      },
      "hardwareSettings": {
        "memory": 1024,
        "hwType": 7
      },
      "installation": {
        "diskSize": 10240
      },
      "orgName": "HQProd",
      "catalogName": "myCatalog",
      "imageName": "CentOS Core"
    }
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: VMware vCloud Director
  account:
    name: My VCD Account
  hardwareSettings:
    memory: 1024
    hwType: 7
  installation:
    diskSize: 10240
  orgName: HQProd
  catalogName: myCatalog
  imageName: CentOS Core
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "VMware vCloud Director",
      "account": {
        "name": "My VCD Account"
      },
      "hardwareSettings": {
        "memory": 1024,
        "hwType": 7
      },
      "installation": {
        "diskSize": 10240
      },
      "orgName": "HQProd",
      "catalogName": "myCatalog",
      "imageName": "CentOS Core"
    }
  ]
}
```

Virtual Targets

citrix-xen

Default builder type: `Citrix Xen Server`

Require Cloud Account: No

This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The Citrix XenServer builder provides information for building XenServer compatible machine images.

The Citrix XenServer builder section has the following definition when using YAML:

```
---
builders:
- type: Citrix Xen Server
       # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Citrix Xen Server",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Citrix Xen Server: `Citrix Xen Server`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. The followi**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Example

The following example shows a Citrix XenServer builder.

If you are using YAML:

```
---
builders:
- type: Citrix Xen Server
  hardwareSettings:
    memory: 1024
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Citrix Xen Server",
      "hardwareSettings": {
        "memory": 1024
      }
    }
  ]
}
```

Hyper-V

Default builder type: `Hyper-V`

Require Cloud Account: No

This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The Hyper-V builder provides information for building Hyper-V compatible machine images. The Hyper-V builder section has the following definition when using YAML:

```
---
builders:
- type: Hyper-V
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Hyper-V",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Hyper-V: `Hyper-V`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. The followi**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).

- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**

  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Example

The following example shows a Hyper-V builder.

If you are using YAML:

```
---
builders:
- type: Hyper-V
  hardwareSettings:
    memory: 1024
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Hyper-V",
      "hardwareSettings": {
        "memory": 1024
      }
    }
  ]
}
```

KVM

Default builder type: `KVM`

Require Cloud Account: No

This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The KVM builder provides information for building KVM (Kernel-based Virtual Machine) compatible machine images. The KVM builder section has the following definition when using YAML:

```
---
builders:
```

```
- type: KVM
       # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "KVM",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for KVM: `KVM`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. The followi**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
- **installation (optional): an object providing low-level installation or first boot options. These override any installation optio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Example

The following example shows a KVM builder.

If you are using YAML:

```
---
builders:
- type: KVM
  hardwareSettings:
    memory: 1024
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "KVM",
      "hardwareSettings": {
        "memory": 1024
      }
    }
  ]
}
```

OVF

Default builder type: `OVF or OVA`

Require Cloud Account: No

The OVF builder provides information for building OVF (Open Virtualization Format) compatible machine images. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The OVF builder section has the following definition when using YAML:

```
---
builders:
- type: OVF or OVA
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "OVF or OVA",
      ...the rest of the definition goes here.
    }
  ]
}
```

The OVF builder has the following valid keys:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for OVF: `OVF or OVA`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. The followi**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
  - `hwType` (optional): an integer providing the hardware type for the machine image. This is the VMware hardware type: 4 (ESXi>3.x), 7 (ESXi>4.x) or 9 (ESXi>5.x)
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**

- `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Example

The following example shows an OVF builder.

If you are using YAML:

```
---
builders:
- type: OVF or OVA
  hardwareSettings:
    memory: 1024
    hwType: 7
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "OVF or OVA",
      "hardwareSettings": {
        "memory": 1024,
        "hwType": 7
      }
    }
  ]
}
```

QCOW2

Default builder type: `QCOW2`

Require Cloud Account: No

The QCOW2 builder provides information for building a QCOW2 compatible machine images. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The QCOW2 builder section has the following definition when using YAML:

```
---
builders:
- type: QCOW2
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "QCOW2",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for QCOW2: `QCOW2`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. The followi**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Example

The following example shows a QCOW2 builder.

If you are using YAML:

```
---
builders:
- type: QCOW2
  hardwareSettings:
    memory: 1024
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "QCOW2",
      "hardwareSettings": {
        "memory": 1024
```

```
      }
    }
  ]
}
```

Raw

Default builder type: `Raw Virtual Disk`

Require Cloud Account: No

The Raw builder provides information for building a Raw Virtual Disk compatible machine images. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The Raw builder section has the following definition when using YAML:

```
---
builders:
- type: Raw Virtual Disk
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Raw Virtual Disk",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for RAW: `Raw Virtual Disk`. To get the available builder type, please refer to *format*
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Example

The following example shows a Raw builder.

If you are using YAML:

```
---
builders:
- type: Raw Virtual Disk
  hardwareSettings:
    memory: 1024
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Raw Virtual Disk",
      "hardwareSettings": {
        "memory": 1024
      }
    }
  ]
}
```

Vagrant Base Box

Default builder type: `Vagrant Base Box`

Require Cloud Account: No

The Vagrant builder provides information for building Vagrant base-box machine images. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The Vagrant builder section has the following definition when using YAML:

```
---
builders:
- type: Vagrant Base Box
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Vagrant Base Box",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Vagrant: `Vagrant Base Box`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. The followi**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.
- `osUser` (optional): a string providing the user used to authenticate to the vagrant base box. This is mandatory if the base box is private, otherwise this value is ignored and the user `vagrant` is used.
- `publicBaseBox` (optional): a boolean determining if the base box to be created is a public base box or not. When public, the os user is vagrant and uses the public (insecure) public key as described in the vagrant documentation
- **`sshKey` (optional): an object providing the public SSH key information to add to the base box. The object contains:**
  - `name` (mandatory): a string providing the name of the public ssh key
  - `publicKey` (mandatory): a string providing the public ssh key. A public key must begin with string `ssh-rsa` or `ssh-dss`. This is mandatory if the base box is private, otherwise this value is ignored and the default public ssh key is used.

Note: You can get copies of the SSH keypairs for public base boxes here.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Examples

Basic Example: Public Base Box

The following example shows a Vagrant builder creating a public base box.

If you are using YAML:

```
---
builders:
- type: Vagrant Base Box
  hardwareSettings:
```

```
    memory: 1024
  publicBaseBox: true
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Vagrant Base Box",
      "hardwareSettings": {
        "memory": 1024
      },
      "publicBaseBox": true
    }
  ]
}
```

Private Base Box Example

The following example shows a Vagrant builder for a private base box (note, that the values used is the same for building a public base box)

If you are using YAML:

```
---
builders:
- type: Vagrant Base Box
  hardwareSettings:
    memory: 1024
  publicBaseBox: false
  osUser: vagrant
  sshKey:
    name: myVagrantPublicKey
    publicKey: ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAABIwAAAQEA6NF8iallvQVp22WDkTkyrtvp9eWW6A8YVr+kz4TjGYe7gHzIw+niNltGEFHzD8+v1I2YJ6
      vagrant insecure public key
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Vagrant Base Box",
      "hardwareSettings": {
        "memory": 1024
      },
      "publicBaseBox": false,
      "osUser": "vagrant",
      "sshKey": {
        "name": "myVagrantPublicKey",
        "publicKey": "ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAABIwAAAQEA6NF8iallvQVp22WDkTkyrtvp9eWW6A8YVr+kz4TjGYe7gHzIw+niNltGEFHzD8+v1I2YJ6
↪vagrant insecure public key"
      }
    }
```

```
  ]
}
```

VirtualBox

Default builder type: `VirtualBox`

Require Cloud Account: No

Oracle VirtualBox

The VirtualBox builder provides information for building Oracle VM VirtualBox compatible machine images. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The VirtualBox builder section has the following definition when using YAML:

```
---
builders:
- type: VirtualBox
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "VirtualBox",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for VirtualBox: `VirtualBox`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. The followi**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Example

The following example shows a VirtualBox builder.

If you are using YAML:

```
---
builders:
- type: VirtualBox
  hardwareSettings:
    memory: 1024
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "VirtualBox",
      "hardwareSettings": {
        "memory": 1024
      }
    }
  ]
}
```

VHD

Default builder type: `VHD`

Require Cloud Account: No

The VHD builder provides information for building VHD (Virtual Hard Disk) compatible machine images. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The VHD builder section has the following definition when using YAML:

```
---
builders:
- type: VHD
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
```

```
      "type": "VHD",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for VHD: `VHD`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. The followi**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Example

The following example shows a VHD builder.

If you are using YAML:

```
---
builders:
- type: VHD
  hardwareSettings:
    memory: 1024
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "VHD",
      "hardwareSettings": {
        "memory": 1024
      }
    }
  ]
}
```

VMware Workstation

Default builder type: `VMware Server`

Require Cloud Account: No

The VMware Workstation builder provides information for building VMware Workstation compatible machine images. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The VMware Workstation builder section has the following definition when using YAML:

```
---
builders:
- type: VMware Server
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "VMware Server",
      ...the rest of the definition goes here.
    }
  ]
}
```

The VMware Workstation builder has the following valid keys:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for VMware Workstation: `VMware Server`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. The followi**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
  - `hwType` (optional): an integer providing the hardware type for the machine image. This is the VMware hardware type: 4 (ESXi>3.x), 7 (ESXi>4.x) or 9 (ESXi>5.x)
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Example

The following example shows a VMware Workstation builder.

If you are using YAML:

```
---
builders:
- type: VMware Server
  hardwareSettings:
    memory: 1024
    hwType: 7
```

If you are using JSON:

```
{
        "builders": [
        {
      "type": "VMware Server",
      "hardwareSettings": {
        "memory": 1024,
        "hwType": 7
      }
        }
]
}
```

VMware vSphere vCenter

Builder type: `VMware vCenter`

Require Cloud Account: Yes

The VMware vCenter builder provides information for building VMware vSphere vCenter compatible machine images. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The VMware vCenter builder section has the following definition when using YAML:

```
---
builders:
- type: VMware vCenter
# the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "VMware vCenter",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for VMware vCenter: `VMware vCenter`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. The followi**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
  - `hwType` (optional): an integer providing the hardware type for the machine image. This is the VMware hardware type: 4 (ESXi>3.x), 7 (ESXi>4.x) or 9 (ESXi>5.x)
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for VMware vCenter: `VMware vCenter`. To get the available builder type, please refer to *format*
- `account` (mandatory): an object providing the VMware vSphere vCenter cloud account information required to publish the built machine image.
- `displayName` (mandatory): a string providing the name of the machine image to display in VMware vSphere vCenter.
- `esxHost` (mandatory): a string providing the esxHost name or ip address.
- `datastore` (mandatory): a string providing the name of the datastore where to store the machine image.
- `network` (optional): a string providing the virtual network name.

vSphere vCenter Cloud Account

Key: `account` Used to authenticate to VMware vSphere vCenter.

The vCenter cloud account has the following valid keys:

- `type` (mandatory): a string providing the cloud account type. Default platform type for VMware vCenter: `VMware vCenter`. To get the available platform type, please refer to *platform*
- `name` (mandatory): a string providing the name of the cloud account. This name can be used in a builder section to reference the rest of the cloud account information.

- `login` (mandatory): a string providing the user name to use to authenticate to the VMware vSphere vCenter platform
- `password` (mandatory): a string providing the password to use to authenticate to the VMware vSphere vCenter platform
- `hostname` (mandatory): a string providing the fully-qualified hostname or IP address of the VMware vSphere vCenter platform.
- `proxyHostname` (optional): a string providing the fully qualified hostname or IP address of the proxy to reach the VMware vSphere vCenter platform.
- `port` (optional): an integer providing the VMware vSphere vCenter platform port number (by default: 443 is used).
- `proxyPort` (optional): an integer providing the proxy port number to use to reach the VMware vSphere vCenter platform.

Note: In the case where `name` or `file` is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Example

The following example shows an vCenter builder with all the information to build and publish a machine image to VMware vSphere vCenter.

If you are using YAML:

```
---
builders:
- type: VMware vCenter
  account:
    type: VMware vCenter
    name: My VCenter account
    login: mylogin
    password: mypassword
    hostname: myhostname
    proxyHostname: myproxyHostname
    proxyPort: '6354'
    port: '443'
  hardwareSettings:
    memory: 1024
    hwType: 7
  installation:
    diskSize: 10240
  esxHost: my_esx_host
  datastore: my_datastore
  displayName: test_Hammr
  network: VM_Network
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "VMware vCenter",
      "account": {
```

```
        "type": "VMware vCenter",
        "name": "My VCenter account",
        "login": "mylogin",
        "password": "mypassword",
        "hostname": "myhostname",
        "proxyHostname": "myproxyHostname",
        "proxyPort": "6354",
        "port": "443"
      },
      "hardwareSettings": {
        "memory": 1024,
        "hwType": 7
      },
      "installation": {
        "diskSize": 10240
      },
      "esxHost": "my_esx_host",
      "datastore": "my_datastore",
      "displayName": "test_Hammr",
      "network": "VM_Network"
    }
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a YAML file `vcenter-account.yml`.

```
---
accounts:
- type: VMware vCenter
  name: My VCenter account
  login: mylogin
  password: mypassword
  hostname: myhostname
  proxyHostname: myproxyHostname
  proxyPort: '6354'
  port: '443'
```

If you are using JSON, create a JSON file `vcenter-account.json`:

```
{
  "accounts": [
    {
      "type": "VMware vCenter",
      "name": "My VCenter account",
      "login": "mylogin",
      "password": "mypassword",
      "hostname": "myhostname",
      "proxyHostname": "myproxyHostname",
      "proxyPort": "6354",
      "port": "443"
    }
  ]
```

```
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: VMware vCenter
  account:
    file: "/home/joris/accounts/vcenter-account.yml"
  hardwareSettings:
    memory: 1024
    hwType: 7
  installation:
    diskSize: 10240
  esxHost: my_esx_host
  datastore: my_datastore
  displayName: test_Hammr
  network: VM_Network
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "VMware vCenter",
      "account": {
        "file": "/home/joris/accounts/vcenter-account.json"
      },
      "hardwareSettings": {
        "memory": 1024,
        "hwType": 7
      },
      "installation": {
        "diskSize": 10240
      },
      "esxHost": "my_esx_host",
      "datastore": "my_datastore",
      "displayName": "test_Hammr",
      "network": "VM_Network"
    }
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: VMware vCenter
  account:
    name: My vCenter Account
  hardwareSettings:
    memory: 1024
    hwType: 7
```

```
  installation:
    diskSize: 10240
  esxHost: my_esx_host
  datastore: my_datastore
  displayName: test_Hammr
  network: VM_Network
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "VMware vCenter",
      "account": {
        "name": "My vCenter Account"
      },
      "hardwareSettings": {
        "memory": 1024,
        "hwType": 7
      },
      "installation": {
        "diskSize": 10240
      },
      "esxHost": "my_esx_host",
      "datastore": "my_datastore",
      "displayName": "test_Hammr",
      "network": "VM_Network"
    }
  ]
}
```

Xen

Default builder type: `Xen`

Require Cloud Account: No

The Xen builder provides information for building a Xen.org compatible machine images. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The Xen builder section has the following definition when using YAML:

```
---
builders:
- type: Xen
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
```

```
      "type": "Xen",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Xen: `Xen`. To get the available builder type, please refer to *format*
- **`hardwareSettings` (mandatory): an object providing hardware settings to be used for the machine image. The followi**
  - `memory` (mandatory): an integer providing the amount of RAM to provide to an instance provisioned from the machine image (in MB).
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - `diskSize` (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Example

The following example shows a Xen builder.

If you are using YAML:

```
---
builders:
- type: Xen
  hardwareSettings:
    memory: 1024
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Xen",
      "hardwareSettings": {
        "memory": 1024
      }
    }
  ]
}
```

Container Targets

Docker

Default builder type: `Docker`

Require Cloud Account: Yes

www.docker.com

The Docker builder provides information for building and publishing the machine image to a Docker Registry. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder type, please refer to *format*

The Docker builder requires cloud account information to upload and register the machine image to a Docker Registry.

The Docker builder section has the following definition when using YAML:

```
---
builders:
- type: Docker
  # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Docker",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Docker: `Docker`. To get the available builder type, please refer to *format*.

Publishing a Machine Image

To publish an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for Docker: `Docker`. To get the available builder type, please refer to *format*.
- `account` (mandatory): an object providing the Docker cloud account information required to publish the built machine image.
- `namespace` (mandatory): a string providing the Docker namespace to publish this machine image to.
- `repositoryName` (mandatory): a string providing the Docker repository name of the image.

- `tagName` (mandatory): a string providing the Docker tag name of the image.

Docker Cloud Account

Key: `account`

Used to authenticate the Docker Registry.

The Docker cloud account has the following valid keys:

- `type` (mandatory): a string providing the cloud account type. Default platform type for Docker is `Docker`. To get the available platform type, please refer to *platform*
- `name` (mandatory): a string providing the name of the cloud account. This name can be used in a builder section to reference the rest of the cloud account information.
- `endpointUrl` (mandatory): a string providing the endpoint URL of the Docker Registry.
- `login` (mandatory): a string providing the login of the user for authenticating to the Docker Registry for publishing images
- `password` (mandatory): a string providing the password of the user for authenticating to the Docker Registry for publishing images
- `file` (optional): a string providing the location of the account information. This can be a pathname (relative or absolute) or an URL.

Note: In the case where `name` or `file` is used to reference a cloud account, all the other keys are no longer required in the account definition for the builder.

Example

The following example shows a Docker builder with all the information to build and publish a machine image to Docker Hub.

If you are using YAML:

```
---
builders:
- type: Docker
  account:
    type: Docker
    name: Docker Hub
    endpointUrl: https://index.docker.io
    login: mylogin
    password: mypassword
  namespace: mylogin
  repositoryName: uforge-image
  tagName: latest
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Docker",
      "account": {
```

```
        "type": "Docker",
        "name": "Docker Hub",
        "endpointUrl": "https://index.docker.io",
        "login": "mylogin",
        "password": "mypassword"
      },
      "namespace": "mylogin",
      "repositoryName": "uforge-image",
      "tagName": "latest"
    }
  ]
}
```

Referencing the Cloud Account

To help with security, the cloud account information can be referenced by the builder section. This example is the same as the previous example but with the account information in another file. Create a YAML file `docker-account.yml`.

```
---
accounts:
- type: Docker
  name: Docker Hub
  endpointUrl: https://index.docker.io
  login: mylogin
  password: mypassword
```

If you are using JSON, create a JSON file `docker-account.json`:

```
{
  "accounts": [
    {
      "type": "Docker",
      "name": "Docker Hub",
      "endpointUrl": "https://index.docker.io",
      "login": "mylogin",
      "password": "mypassword"
    }
  ]
}
```

The builder section can either reference by using `file` or `name`.

Reference by file:

If you are using YAML:

```
---
builders:
- type: Docker
  account:
    file: "/path/to/docker-account.yml"
  namespace: mylogin
  repositoryName: uforge-image
  tagName: latest
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Docker",
      "account": {
            "file": "/path/to/docker-account.json"
      },
      "namespace": "mylogin",
      "repositoryName": "uforge-image",
      "tagName": "latest"
    }
  ]
}
```

Reference by name, note the cloud account must already be created by using `account create`.

If you are using YAML:

```
---
builders:
- type: Docker
  account:
    name: Docker Hub
  namespace: mylogin
  repositoryName: uforge-image
  tagName: latest
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "Docker",
      "account": {
        "name": "Docker Hub"
        },
      "namespace": "mylogin",
      "repositoryName": "uforge-image",
      "tagName": "latest"
    }
  ]
}
```

LXC

Default builder type: `LXC Container`

Require Cloud Account: No

www.linuxcontainers.org

The LXC builder provides information for building a Linux container image. This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder type, please refer to *format*

The LXC builder section has the following definition when using YAML:

```
---
builders:
- type: LXC Container
  # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "LXC Container",
      ...the rest of the definition goes here.
    }
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for LXC: `LXC Container`. To get the available builder type, please refer to *format*.

Physical Targets

ISO

Default builder type: `ISO`

Require Cloud Account: No

This builder type is the default name provided by UForge AppCenter.

Note: This builder type name can be changed by your UForge administrator. To get the available builder types, please refer to *format*

The ISO builder provides information for building ISO images.

The ISO builder section has the following definition when using YAML:

```
---
builders:
- type: ISO
        # the rest of the definition goes here.
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "ISO",
      ...the rest of the definition goes here.
    }
```

```
  ]
}
```

Building a Machine Image

For building an image, the valid keys are:

- `type` (mandatory): a string providing the machine image type to build. Default builder type for ISO: `ISO`. To get the available builder type, please refer to *format*
- **`installation` (optional): an object providing low-level installation or first boot options. These override any installatio**
  - diskSize (mandatory): an integer providing the disk size of the machine image to create. Note, this overrides any disk size information in the stack. This cannot be used if an advanced partitioning table is defined in the stack.

Note: When building from a scan, your yaml or json file must contain an `installation` section in `builders`. This is mandatory when you create a new template, but might be missing when you build from a scan. Make sure it is present or your build will fail.

Example

The following example shows an ISO builder.

If you are using YAML:

```
---
builders:
- type: ISO
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "ISO"
    }
  ]
}
```

CHAPTER 11

Migrating a Live System

Hammr allows you to migrate a live system. The key steps in migrating your system are:

1. Scan your system, this sends a scan report back to your UForge account
2. From the scan report, build and publish a machine image
3. Finally provision an instance from the published machine image (effectively migrating the system)

Optionally, at step #2, you can import the scan report to create a template. This allows you to change the content prior to building a machine image.

First, scan the system you wish to migrate by running `scan run`. This "deep scans" the live system, reporting back the meta-data of every file and package that makes up the running workload. The following is an example of a scan of a live system:

Note: The following example shows a simple scan (without overlay). If you would like the overlay, add `--overlay` argument to the command.

```
$ hammr scan run --ip 192.0.2.0 --scan-login LOGIN --name scan-name
Password for root@192.0.2.0:
... uforge-scan v2.54 (Feb 18 2014 13:16:37) (SVN Revision: 21664)
... Distribution:        Debian / 6.0.9 / x86_64
... Current System Name: Linux
...          Node Name:   test-deb-1-0rev2-vbox
...          Release:     2.6.32-5-amd64
...          Version:     #1 SMP Tue May 13 16:34:35 UTC 2014
...          Machine:     x86_64
...          Domain:      (none)
... Server URL: http://192.168.10.141/ufws-3.3
... User: root
... Testing connection to the service...
...                                    SUCCESS!
...
...
Searching scan on uforge ...
```

```
|>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| 100%: Successfully scanned |<<<<<<<<<<<<<<<<<<<<<<<<
↪<<<<<<<<<<|
OK: Scan successfully
```

Once you have run the scan of your system, a scan report is saved to your account. You can list your scans by running `scan list`. The output will be similar to the following. As you can see below, the "scanExample" is the group name. The actual scan appears below it with "Scan #1" added to the name. If you run the scan on the same machine again, the scan number will increase. This allows you to compare scans.

Note: In this example, "scanExample" is a simple scan. If it was a scan with overlay an "X" will appear in the column "With overlay" for the group name.

```
$ hammr scan list
Getting scans for [root] ...
+-----+------------------------------+--------+-----------------+--------------+
| Id  |             Name             | Status |  Distribution   | With overlay |
+=====+==============================+========+=================+==============+
| 133 | scanExample                  |        | Debian 6 x86_64 |              |
+-----+------------------------------+--------+-----------------+--------------+
| 149 |         scanExample Scan #1  | Done   |                 |              |
+-----+------------------------------+--------+-----------------+--------------+
Found 1 scans
```

If you are simply moving your system from one cloud provider to another, you can then simply build a machine image from this scan by running `scan build`. The following is an example which builds a machine image from a scan:

```
$ hammr scan build --id 192 --file openstack-builder.yml
OK: Syntax of template file [openstack-builder.yml] is ok
Generating 'openstack' image (1/1)
|>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| 100%: Done, created on ... |<<<<<<<<<<<<<<<<<<<<<<<<
↪<<<<<<<<<<|
OK: Generation 'openstack' ok
```

In the example above, you will need to have a YAML file which defines the `builder` parameters for the type of machine image you want to create. This is NOT a full template configuration file, but just the builders parameters. For example:

```
---
builders:
- type: openstack
  hardwareSettings:
    memory: 1024
  installation:
    diskSize: 2000
  account: Openstack OW2
  tenant: opencloudware
  imageName: scan-test
  publicImage: 'no'
```

If you are using JSON:

```
{
  "builders": [
    {
      "type": "openstack",
```

```
      "hardwareSettings": {
        "memory": 1024
      },
      "installation": {
        "diskSize": 2000
      },
      "account": "Openstack OW2",
      "tenant": "opencloudware",
      "imageName": "scan-test",
      "publicImage": "no"
    }
  ]
}
```

Updating a Template Before Migrating

Hammr also allows you to modify or update packages that are part of the system you want to migrate. To do this, you first need to transform the scan report to a template. You can then modify any part of this new template prior to building the final machine image used for migration.

To create a template from your scan you will need to run `scan import`. The following is an example that shows a scan conversion to a template within UForge.

```
$ hammr scan import --id 123 --name "MyScan" --version "1.0"
Import scan id [123] ...
|>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>| 100%: Imported 28 May 2... |<<<<<<<<<<<<<<<<<<<<<<
↪<<<<<<<<<<|
OK: Importing ok
```

Once this template is created, you can now update it. In this release, hammr does not provide a mechanism to update existing templates. So to update a template you must:

1. Export the template – see section *Importing and Exporting* for more information.
2. Extract the archive, retrieving the configuration file (JSON or YAML).
3. Update the configuration file (JSON or YAML) with the required changes, you will need to change either the template name or version so you do not get a conflict when you create the new template.
4. Create a new template – see section *Creating and Managing Templates*.
5. Build and publish the machine image (which effectively migrates the workload with the changes) – see section *Building and Publishing Machine Images*

CHAPTER 12

Changelog

hammr 3.7.5 (2017-06-12)

Evolutions:

- Modification of VMware VCenter image publication
- Support restrictions on Software Bundle

Bug fixes:

- Version check when using hammr commands without interactive mode
- A scan name including a space cannot be specified with hammr scan run
- hammr image list displays 0B for images generated from a scan

Compatibility with UForge AppCenter 3.7.fp5-1 only

hammr 3.7.4 (2017-28-04)

Evolutions:

- Compatibility with UForge AppCenter 3.7.fp4-1 only
- Add overlay argument to scan run command to run a scan with overlay

Bug fixes:

- Flag acceptAutoSigned not working for image download
- Name for the builder account can now be read from an external file
- Fixes on documentation

hammr 3.7.3 (2017-21-03)

Evolutions:

- Compatibility with UForge AppCenter 3.7.fp3-1 only

Bug fixes:

- Improve documentation for install compatibility between Hammr and UForge

hammr 3.7-3 (2017-16-02)

Evolutions:

- Compatibility with UForge AppCenter 3.7-3 only
- Align bundle specification with UForge
- Support YAML files as input

Bug fixes:

- Improve documentation for install compatibility between Hammr and UForge

hammr 3.7.2-1 (2017-14-02)

Evolutions:

- Compatibility with UForge AppCenter 3.7.fp2-1 only
- Add Azure Resource Manager publish support
- Add Docker publish support
- Modify documentation for multi-nics option
- Align bundle specification with UForge
- Support YAML files as input

Bug fixes:

- Improve documentation for install compatibility between Hammr and UForge

hammr 3.7-2 (2017-31-01)

Evolutions:

- Compatibility with UForge AppCenter 3.7-2 only
- Improve release process for Hammr
- Add Fujitsu K5 publish support
- Add release notes in documentation

Bug fixes:

- Fixes on documentation

hammr-3.6 1.1 (2016-16-12)

Evolutions:

- Improve project setup.py clean command
- Add travis CI build for the project
- Add an optional parameter to allow to change the ssh port used to connect on the running machine
- Ability to use a directory as source for bundle

Bug fixes:

- Scan build method generate exception
- Fix typo in os help message
- Some fixes on documentation
- A name including a space cannot be specified with hammr template clone
- Account list gives the class name instead of the account type
- The usage of the pkg parameter of hammr os search is not correct

hammr-3.6 0.1 (2016-07-01)

Evolutions:

- **Compatibility with UForge AppCenter 3.6**
  - Target formats and target platforms support
  - Builder part has been updated
- Hammr documentation now inside github repository
- Improve setup.py clean command
- Hammr uses a new download utility

Bug fixes:

- Ability to specify a timezone inside “updateTo” field for “stack”

Known issues:

- Amazon AWS format is not working
- Bootscript order is mandatory (incompatibility with Hammr on UForge AppCenter 3.5.1)
- Not possible to use both hammr 0.2.x and hammr-3.6 on the same system

0.2.5.10 (2016-04-29)

Evolutions:

- Added hammr documentation to the github project
- Add support for uforge-python-sdk 3.5.1.4: ability to do streaming download

Bug fixes:

- `hammr scan run` fails when searching scan on uforge
- Using a relative path to the json file seems to invoke an error
- hammr image publish returns exception if there is no cloud account

0.2.5.9 (2015-12-18)

Evolutions:

- Add compatibility with Outscale format

0.2.5.8 (2015-11-20)

Evolutions:

- Increase timeout value

Bug fixes:

- Cannot install hammr because of a dependency error (issue #45)

0.2.5.7 (2015-09-21)

Evolutions:

- Reuse existing bundles option while importing templates (issue #26)
- Template export directory clean up (issue #43)

0.2.5.6 (2015-08-29)

Bug fixes:

- Fix issue #38 - Could be nice to have a way to specify credentials file from command line
- Fix issue #31 - “hammr scan delete” deletes every scan if scan id and scan instance id is the same.

0.2.5.5 (2015-08-04)

Evolutions:

- Add support for lxc and targz for Hammr

Bug fixes:

- Fix issue #34 - Exit status of Hammr command
- Enhance the error message if an issue occurs when trying to download a machine image

CHAPTER 13

Trademarks

UForge is a registered trademark of UShareSoft, a Fujitsu company.

LINUX is a registered trademark of Linus Torvalds.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Oracle, GlassFish, Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle Corporation and/or its affiliates.

Apache Ant, Ant, and Apache are trademarks of The Apache Software Foundation.

UNIX is a registered trademark of the Open Group in the United States and in other countries.

Other company names and product names are trademarks or registered trademarks of their respective owners.