Docker

Module for installing and managing docker

**Latest version is compatible with:**

* Puppet Enterprise 2016.4.x, 2016.2.x, 2016.1.x, 2015.3.x, 2015.2.x, 3.8.x, 3.7.x, 3.3.x, 3.2.x
* Puppet >= 3.4.0
  + RedHat,

* + Ubuntu,

* + Debian,

* + CentOS,

* + Archlinux,

* + Gentoo,

* + Fedora

To use this module, add this declaration to your Puppetfile:

mod 'garethr-docker', '5.3.0'

To manually install this module with puppet module tool:

puppet module install garethr-docker --version 5.3.0

**Support**

This module is currently tested on:

* Debian 8.0
* Debian 7.8
* Ubuntu 12.04
* Ubuntu 14.04
* Centos 7.0
* Centos 6.6

It may work on other distros and additional operating systems will be supported in the future. It's definitely been used with the following too:

* Archlinux
* Amazon Linux
* Fedora
* Gentoo

Examples

* [Launch vNext app in Docker using Puppet](https://github.com/garethr/puppet-docker-vnext-example) This example contains a fairly simple example using Vagrant to launch a Linux virtual machine, then Puppet to install Docker, build an image and run a container. For added spice the container runs a ASP.NET vNext application.
* [Multihost containers connected with Consul](https://github.com/garethr/puppet-docker-example) Launch multiple hosts running simple application containers and connect them together using Nginx updated by Consul and Puppet.
* [Configure Docker Swarm using Puppet](https://github.com/garethr/puppet-docker-swarm-example) Build a cluster of hosts running Docker Swarm configured by Puppet.

**Usage**

The module includes a single class:

**include** 'docker'

By default this sets up the docker hosted repository if necessary for your OS and installs the docker package and on Ubuntu, any required Kernel extensions.

If you don't want this module to mess about with your Kernel then you can disable this feature like so. It is only enabled (and supported) by default on Ubuntu:

**class** { 'docker':

manage\_kernel => false,

}

If you want to configure your package sources independently, inform this module to not auto-include upstream sources (This is already disabled on Archlinux as there is no further upstream):

**class** { 'docker':

use\_upstream\_package\_source => false,

}

Docker recently [launched new official repositories](http://blog.docker.com/2015/07/new-apt-and-yum-repos/#comment-247448) which are now the default for the module from version 5. If you want to stick with the old respoitories you can do so with the following:

**class** { 'docker':

package\_name => 'lxc-docker',

package\_source\_location => 'https://get.docker.com/ubuntu',

package\_key\_source => 'https://get.docker.com/gpg',

package\_key => '36A1D7869245C8950F966E92D8576A8BA88D21E',

package\_release => 'docker',

}

Docker also provide a [commercially supported](https://docs.docker.com/docker-trusted-registry/install/install-csengine/) version of the Docker Engine, called Docker CS, available from a separate repository. This can be installed with the module using the following:

**class** { 'docker':

docker\_cs => true,

}

The module also now uses the upstream repositories by default for RHEL based distros, including Fedora. If you want to stick with the distro packages you should use the following:

**class** { 'docker':

use\_upstream\_package\_source => false,

package\_name => 'docker',

}

By default the docker daemon will bind to a unix socket at /var/run/docker.sock. This can be changed, as well as binding to a tcp socket if required.

**class** { 'docker':

tcp\_bind => ['tcp://127.0.0.1:4243','tcp://10.0.0.1:4243'],

socket\_bind => 'unix:///var/run/docker.sock',

ip\_forward => true,

iptables => true,

ip\_masq => true,

bridge => **br0**,

fixed\_cidr => '10.20.1.0/24',

default\_gateway => '10.20.0.1',

}

For TLS setup you should upload related files (such as CA certificate, server certificate and key) and use their paths in manifest

**class** { 'docker':

tcp\_bind => ['tcp://0.0.0.0:2376'],

tls\_enable => true,

tls\_cacert => '/etc/docker/tls/ca.pem',

tls\_cert => '/etc/docker/tls/cert.pem',

tls\_key => '/etc/docker/tls/key.pem',

}

Unless specified this installs the latest version of docker from the docker repository on first run. However if you want to specify a specific version you can do so, unless you are using Archlinux which only supports the latest release. Note that this relies on a package with that version existing in the reposiroty.

**class** { 'docker':

version => '0.5.5',

}

And if you want to install a specific rpm package of docker you can do so:

**class** { 'docker' :

manage\_package => true,

use\_upstream\_package\_source => false,

package\_name => 'docker-engine'

package\_source => 'https://get.docker.com/rpm/1.7.0/centos-6/RPMS/x86\_64/docker-engine-1.7.0-1.el6.x86\_64.rpm',

prerequired\_packages => [ 'glibc.i686', 'glibc.x86\_64', 'sqlite.i686', 'sqlite.x86\_64', 'device-mapper', 'device-mapper-libs', 'device-mapper-event-libs', 'device-mapper-event' ]

}

And if you want to track the latest version you can do so:

**class** { 'docker':

version => 'latest',

}

In some cases dns resolution won't work well in the container unless you give a dns server to the docker daemon like this:

**class** { 'docker':

dns => '8.8.8.8',

}

To add users to the Docker group you can pass an array like this:

**class** { 'docker':

docker\_users => ['user1', 'user2'],

}

To add daemon labels you can pass an array like this:

**class** { 'docker':

labels => ['storage=ssd','stage=production'],

}

The class contains lots of other options, please see the inline code documentation for the full options.

### 

### Images

The next step is probably to install a docker image; for this we have a defined type which can be used like so:

**docker::image** { 'base': }

This is equivalent to running docker pull base. This is downloading a large binary so on first run can take a while. For that reason this define turns off the default 5 minute timeout for exec. Takes an optional parameter for installing image tags that is the equivalent to running docker pull -t="precise" ubuntu:

**docker::image** { 'ubuntu':

image\_tag => 'precise'

}

Note: images will only install if an image of that name does not already exist.

A images can also be added/build from a dockerfile with the docker\_file property, this equivalent to running

 docker build -t ubuntu - < /tmp/Dockerfile

**docker::image** { 'ubuntu':

docker\_file => '/tmp/Dockerfile'

}

Images can also be added/build from a directory containing a dockerfile with the docker\_dir property, this is equivalent to running

docker build -t ubuntu /tmp/ubuntu\_image

**docker::image** { 'ubuntu':

docker\_dir => '/tmp/ubuntu\_image'

}

You can trigger a rebuild of the image by subscribing to external events like Dockerfile changes:

**docker::image** { 'ubuntu':

docker\_file => '/tmp/Dockerfile'

subscribe => **File**['/tmp/Dockerfile'],

}

**file** { '/tmp/Dockerfile':

ensure => **file**,

source => 'puppet:///modules/someModule/Dockerfile',

}

You can also remove images you no longer need with:

**docker::image** { 'base':

ensure => 'absent'

}

**docker::image** { 'ubuntu':

ensure => 'absent',

image\_tag => 'precise'

}

If using hiera, there's a docker::images class you can configure, for example:

---

classes:

- docker::images

docker::images::images:

ubuntu:

image\_tag: 'precise'

### Containers

Now you have an image you can launch containers:

**docker::run** { 'helloworld':

image => 'base',

command => '/bin/sh -c "while true; do echo hello world; sleep 1; done"',

}

This is equivalent to running the following:

**docker** **run** **-d** **base** **/bin/sh** **-c** "while true; do echo hello world; sleep 1; done"

This will launch a Docker container managed by the local init system.

Run also takes a number of optional parameters:

**docker::run** { 'helloworld':

image => 'base',

command => '/bin/sh -c "while true; do echo hello world; sleep 1; done"',

ports => ['4444', '4555'],

expose => ['4666', '4777'],

links => ['mysql:db'],

net => 'my-user-def-net',

volumes => ['/var/lib/couchdb', '/var/log'],

volumes\_from => '6446ea52fbc9',

memory\_limit => '10m', *# (format: '<number><unit>', where unit = b, k, m or g)*

cpuset => ['0', '3'],

username => 'example',

hostname => 'example.com',

env => ['FOO=BAR', 'FOO2=BAR2'],

env\_file => ['/etc/foo', '/etc/bar'],

dns => ['8.8.8.8', '8.8.4.4'],

restart\_service => true,

privileged => false,

pull\_on\_start => false,

before\_stop => 'echo "So Long, and Thanks for All the Fish"',

after => [ 'container\_b', 'mysql' ],

depends => [ 'container\_a', 'postgres' ],

extra\_parameters => [ '--restart=always' ],

}

Ports, expose, env, env\_file, dns and volumes can be set with either a single string or as above with an array of values.

Specifying pull\_on\_start will pull the image before each time it is started.

Specifying before\_stop will execute a command before stopping the container.

The after option allows expressing containers that must be started before. This affects the generation of the init.d/systemd script.

The depends option allows expressing container dependencies. The depended container will be started before this container(s), and this container will be stopped before the depended container(s). This affects the generation of the init.d/systemd script. You can use depend\_services to specify dependency for generic services (non-docker) that should be started before this container.

extra\_parameters : An array of additional command line arguments to pass to the docker run command. Useful for adding additional new or experimental options that the module does not yet support.

The service file created for systemd based systems enables automatic restarting of the service on failure by default.

To use an image tag just append the tag name to the image name separated by a semicolon:

**docker::run** { 'helloworld':

image => 'ubuntu:precise',

command => '/bin/sh -c "while true; do echo hello world; sleep 1; done"',

}

By default the generated init scripts will remove the container (but not any associated volumes) when the service is stopped or started. This behaviour can be modified using the following, with defaults shown:

**docker::run** { 'helloworld':

remove\_container\_on\_start => true,

remove\_volume\_on\_start => false,

remove\_container\_on\_stop => true,

remove\_volume\_on\_stop => false,

}

If using hiera, there's a docker::run\_instance class you can configure, for example:

---

classes:

- docker::run\_instance

docker::run\_instance::instance:

helloworld:

image: 'ubuntu:precise'

command: '/bin/sh -c "while true; do echo hello world; sleep 1; done"'

### Networks

As of Docker 1.9.x, Docker has official support for networks. The module now exposes a type, docker\_network, used to manage those. This works like:

**docker\_network** { 'my-net':

ensure => **present**,

driver => 'overlay',

subnet => '192.168.1.0/24',

gateway => '192.168.1.1',

ip\_range => '192.168.1.4/32',

}

Only the name is required, along with an ensure value. If you don't pass a driver Docker network will use the default bridge. Note that some networks require the Docker daemon to be configured to use them, for instance for the overlay network you'll need a cluster store configured. You can do that on the docker class like so:

extra\_parameters => '--cluster-store=<backend>://172.17.8.101:<port> --cluster-advertise=<interface>:2376'

If using hiera, there's a docker::networks class you can configure, for example:

---

classes:

- docker::networks

docker::networks::networks:

local-docker:

ensure: 'present'

subnet: '192.168.1.0/24'

gateway: '192.168.1.1'

The network defined can then be used on a docker::run resource with the net parameter.

### 

### Compose

Docker Compose allows for describing a set of containers in a simple YAML format, and then running a command to build and run those containers.

The docker\_compose type included in the module allows for using Puppet to run Compose. This means you can have Puppet remediate any issues and make sure reality matches the model in your Compose file.

Before using the docker\_compose type make sure the docker-compose utility is installed:

**class** {'docker::compose':

ensure => **present**,

}

Here's an example. Given the following Compose file:

compose\_test:

image: ubuntu:14.04

command: /bin/sh -c "while true; do echo hello world; sleep 1; done"

That could be added to the machine you're running Puppet using a file resource or any other means.

Then define a docker\_compose resource pointing at the Compose file like so:

**docker\_compose** { '/tmp/docker-compose.yml':

ensure => **present**,

}

Now when Puppet runs it will automatically run Compose is required, for example because the relevant Compose services aren't running.

You can also pass additional options (for example to enable experimental features) as well as provide scaling rules. The following example requests 2 containers be running for example. Puppet will now run Compose if the number of containers for a given service don't match the provided scale values.

**docker\_compose** { '/tmp/docker-compose.yml':

ensure => **present**,

scale => {

'compose\_test' => 2,

},

options => '--x-networking'

}

It is also possible to give options to the docker-compose up command such as --remove-orphans using the up\_args option.

### Private registries

By default images will be pushed and pulled from [index.docker.io](http://index.docker.io/) unless you've specified a server. If you have your own private registry without authentication, you can fully qualify your image name. If your private registry requires authentication you may configure a registry:

**docker::registry** { 'example.docker.io:5000':

username => 'user',

password => 'secret',

email => 'user@example.com',

}

You can logout of a registry if it is no longer required.

**docker::registry** { 'example.docker.io:5000':

ensure => 'absent',

}

If using hiera, there's a docker::registry\_auth class you can configure, for example:

docker::registry\_auth::registries:

'example.docker.io:5000':

username: 'user1'

password: 'secret'

email: 'user1@example.io'

### Exec

Docker also supports running arbitrary commands within the context of a running container. And now so does the Puppet module.

**docker::exec** { 'cron\_allow\_root':

detach => true,

container => 'mycontainer',

command => '/bin/echo root >> /usr/lib/cron/cron.allow',

tty => true,

unless => 'grep root /usr/lib/cron/cron.allow 2>/dev/null',

}