redock

Release 0.5.8

Sep 27, 2017

Contents

Welcome to the documentation for Redock 0.5.8. Redock is a human friendly wrapper around Docker, the Linux container engine. Docker implements a lightweight form of virtualization that makes it possible to start and stop "virtual machines" in less than a second. Redock comes in two parts:

1. The command line program `redock` whose main goal to to be simple to use.
2. A Python API for more advanced use cases (for example the command line program is built on top of the API).

The documentation below also consists of two parts: The readme with installation and usage instructions and the API documentation.

CHAPTER 1

Introduction & usage

The first part of the documentation is the readme which is targeted at users of the `redock` command line program. Here are the topics discussed in the readme:

redock: A human friendly wrapper around Docker

Redock is a human friendly wrapper around Docker, the Linux container engine. Docker implements a lightweight form of virtualization that makes it possible to start and stop "virtual machines" in less than a second. Before Docker the use of virtualization meant conventional virtual machines with all of the associated bloat. Docker makes it ridiculously cheap and fast to start/save/kill containers. This opens up exciting new possibilities for DevOps:

- Complex build environments can be split up into isolated containers where each container is concerned with the build requirements of a single project. If a project's build goes out of hand you just trash the container and go on your merry way :-)
- The correctness of automated deployment systems (and distributed systems in general) can be verified by using containers to host the configuration management server and agents.
- To be honest, Docker is so fast that I could imagine myself building a test suite of a complete cluster on top of it.

The last point is the reason why I started working on Redock. In my initial experiments with Docker I found a lot of sharp edges (both in the lack of documentation and in the implementation of Docker and its Python API) but at the same time my fingers were itching to wrap Docker in an easy to use and human friendly wrapper to try and unleash its potential.

What Redock gives you is Docker without all the hassle: When you create a container, Redock will install, configure and start an SSH server and open an interactive SSH session to the container. What you do with the container after that is up to you...

Status

Redock should be considered alpha quality software. So far it has been used by a single person (me). Right now it's intended for development work, not production use. This might change over time, depending on my experiences with Docker over the coming weeks / months (I'm specifically concerned with stability and performance).

By the way the same can and should be said about Docker (its site says as much). During heavy testing of Redock I've experienced a number of unhandled kernel mode NULL pointer dereferences that didn't crash the host system but certainly also didn't inspire confidence ;-). It should be noted that these issues didn't occur during regular usage; only heavy testing involving the creation and destruction of dozens of Docker containers would trigger the issue.

There's one thing I should probably mention here as a disclaimer: Redock rewrites your SSH configuration (`~/.ssh/config`) using update-dotdee. I've tested this a fair bit, but it's always a good idea to keep backups (hint).

I'm currently using Redock on Ubuntu 12.04 with Docker 0.6.3 and a Linux 3.8.0 kernel (as suggested in Docker's installation instructions for Ubuntu). I've only just switched to these versions and it seems they may have solved the stability issues I mentioned above (time will tell :-).

Usage

You will need to have Docker installed before you can use Redock, please refer to Docker's installation instructions. You may also want to add your user account to the `docker` group so you can connect to Docker without `sudo` (this took me a while to realize when I switched to Docker 0.6 :-). After you've installed Docker you can install Redock using the following command:

```
$ pip install redock
```

This downloads and installs Redock using pip (the Python package manager). Redock is written in Python so you need to have Python installed. Redock pulls in a bunch of dependencies so if you're familiar with virtual environments you might want to use one :-). Once you've installed Docker and Redock, here's how you create a container:

```
$ redock start test
```

If you run this command interactively and you start a single container, Redock will start an interactive SSH session that connects you to the container. In any case you will now be able to connect to the container over SSH using the name you gave to the container suffixed with `-container`:

```
$ ssh test-container
```

This works because your `~/.ssh/config` has been updated to include a host definition for the container. This means you can connect using rsync or anything else which works on top of SSH (e.g. to bootstrap a configuration management system). When you're done playing around with the container you can save your changes with the following command:

```
$ redock commit test
```

This command will persist the state of the container's file system in a Docker image. The next time you run Redock with the same name it will create a container based on the existing disk image. To kill and delete a running container you use the following command:

```
$ redock kill test
```

This will discard all changes made to the file system inside the container since the last time that `redock commit` was used. The Docker image associated with a container can be deleted like this:

```
$ redock delete test
```

Naming conventions

In the examples above the name `test` is used. This name is used by Redock to identify the running container (created with `redock start`) and any associated images (created with `redock commit`). By using multiple names you can run multiple containers in parallel and you can suspend / resume "long term" containers.

The names accepted by Redock are expected to be of the form `repository:tag` (two words separated by a colon):

1. The first part (`repository` in the example) is a top level name space for Docker images. For example there is a repository called `ubuntu` that contains the official base images. Similarly Redock uses the repository `redock` for the base image it creates on the first run.
2. The second part (`tag` in the example) is the name of a specific container and/or image; I usually just sets it to the host name of the system that will be running inside the container.

If the colon is missing the `repository` will be set to your username (based on the environment variable `$USER`).

Contact

The latest version of Redock is available on PyPI and GitHub. The API documentation is hosted on Read The Docs. For bug reports please create an issue on GitHub. If you have questions, suggestions, etc. feel free to send me an e-mail at peter@peterodding.com.

License

This software is licensed under the MIT license.

© 2013 Peter Odding.

CHAPTER 2

API documentation

The second part of the documentation is targeted at developers who wish to use Redock in their own Python projects. Here are the contents of the API documentation:

Documentation for the Redock API

Contents

Main Redock API

The `redock.api` module defines two classes and two exception types:

- `Container`
- `Image`
- `NoContainerRunning`
- `SecureShellTimeout`

class `redock.api.`**`Container`**(*image*, *hostname=None*, *timeout=10*)

`Container` is the main entry point to the Redock API. It aims to provide a simple to use representation of

Docker containers (and in extension Docker images). You'll probably never need most of the methods defined in this class; if you're getting started with Redock you should focus on these methods:

- *Container.start()*
- *Container.commit()*
- *Container.kill()*

After you create and start a container with Redock you can do with the container what you want by starting out with an SSH connection. When you're done you either save your changes or discard them and kill the container. That's probably all you need from Redock :-)

__init__(*image*, *hostname=None*, *timeout=10*)
Initialize a *Container* from the given arguments.

Parameters

- **image** – The repository and tag of the container's image (in the format expected by *Image.coerce()*).
- **hostname** – The host name to use inside the container. If none is given, the image's tag is used.
- **timeout** – The timeout in seconds while waiting for a container to become reachable over SSH (a couple of seconds should be plenty).

check_active()
Check if the *Container* is associated with a running Docker container. If no running Docker container is found, *NoContainerRunning* is raised.

commit(*message=None*, *author=None*)
Commit any changes to the running container to the associated image. Corresponds to the `docker commit` command.

Raises *NoContainerRunning* if an associated Docker container is not already running.

Parameters

- **message** – A short message describing the commit (a string).
- **author** – The name of the author (a string).

delete()
Delete the image associated with the container (if any). The data in the image will be lost.

expand_id(*short_id*, *candidate_ids*)
`docker.Client.create_container()` and `docker.Client.commit()` report short ids (12 characters) while `docker.Client.containers()` and `docker.Client.images()` report long ids (65 characters). I'd rather use the full ids where possible. This method translates short ids into long ids at the expense of an additional API call (who cares).

Raises `exceptions.Exception` if no long id corresponding to the short id can be matched (this might well be a purely theoretical problem, it certainly shouldn't happen during regular use).

Parameters

- **short_id** – A short id of 12 characters.
- **candidate_ids** – A list of available long ids.

Returns The long id corresponding to the given short id.

find_container()
Check to see if the current *Container* has an associated Docker container that is currently running.

Returns `True` when a running container exists, `False` otherwise.

find_image(*image_to_find*)
Find the most recent Docker image with the given repository and tag.

Parameters **image_to_find** – The *Image* we're looking for.

Returns The most recent *Image* available, or `None` if no images were matched.

get_ssh_client_command(*ip_address=None*, *port_number=None*)
Generate an SSH client command line that connects to the container (assumed to be running).

Parameters

- **ip_address** – This optional argument overrides the default IP address (which is otherwise automatically discovered).
- **port_number** – This optional argument overrides the default port number (which is otherwise automatically discovered).

Returns The SSH client command line as a list of strings containing the command and its arguments.

kill()
Kill and remove the container. All changes since the last time that *Container.commit()* was called will be lost.

revoke_ssh_access()
Remove the container's SSH client configuration from `~/.ssh/config`.

setup_ssh_access()
Update `~/.ssh/config` to make it easy to connect to the container over SSH from the host system. This generates a host definition to include in the SSH client configuration file and uses update-dotdee to merge the generated host definition with the user's existing SSH client configuration file.

ssh_alias
Get the SSH alias that should be used to connect to the container.

ssh_config_file
Get the pathname of the SSH client configuration for the container.

ssh_endpoint
Wait for the container to become reachable over SSH and get a tuple with the IP address and port number that can be used to connect to the container over SSH.

start()
Create and start the Docker container. On the first run of Redock this creates a base image using *redock.base.create_base_image()*.

start_supervisor()
Starts the container and runs Supervisor inside the container.

class `redock.api.`**Image**(*repository*, *tag*, *id=None*)
Simple representation of Docker images.

__init__(*repository*, *tag*, *id=None*)
Initialize an *Image* instance from the given arguments.

Parameters

- **repository** – The name of the image's repository.
- **tag** – The image's tag (name).
- **id** – The unique hash of the image (optional).

static coerce(*value*)
Coerce strings to *Image* objects.

Raises exceptions.ValueError when a string with an incorrect format is given.

Parameters value – The name of the image, expected to be a string of the form repository:tag. If an *Image* object is given it is returned unmodified.

Returns An *Image* object.

key
Get a tuple with the image's repository and tag.

name
Get the human readable name of an *Image* as a string of the form repository:tag.

unique_name
Get the machine readable unique name of an *Image*. If the image has a unique hash that will be used, otherwise a string of the form repository:tag is returned.

exception redock.api.**NoContainerRunning**
Raised by *Container.check_active()* when a *Container* doesn't have an associated Docker container running.

exception redock.api.**SecureShellTimeout**
Raised by *Container.ssh_endpoint* when Redock fails to connect to the Docker container within a reasonable amount of time (10 seconds by default).

class redock.api.**Session**
Dumb object to hold session variables associated with a running Docker container.

__init__()

reset()
Reset all known session variables to None.

Base image handling API

The *redock.base* module implements the initialization of the base image used by Redock. You'll probably never need to use this module directly because *redock.api.Container.start()* calls *find_base_image()* and *create_base_image()* as needed.

redock.base.**create_base_image**(*client*)
Create the base image that's used by Redock to create new containers. This base image differs from the ubuntu:precise image (on which it is based) on a couple of points:

- Automatic installation of recommended packages is disabled to conserve disk space.
- The Ubuntu package mirror is set to a geographically close location to speed up downloading of system packages (see *redock.utils.select_ubuntu_mirror()*).
- The package list is updated to make sure apt-get installs the most up to date packages.
- The following system packages are installed:

  language-pack-en-base In a base Docker Ubuntu 12.04 image lots of commands complain loudly about the locale. This silences the warnings by fixing the problem (if you want to call it that).

  openssh-server After creating a new container Redock will connect to it over SSH, so having an SSH server installed is a good start :-)

supervisor The base Docker Ubuntu 12.04 image has init (upstart) disabled. Indeed we don't need all of the bagage that comes with init but it is nice to have a process runner for the SSH server (and eventually maybe more).

- The initscripts and upstart system packages are marked 'on hold' so that apt-get will not upgrade them. This makes it possible to run `apt-get dist-upgrade` inside containers.
- An SSH key pair is generated and the SSH public key is installed inside the base image so that Redock can connect to the container over SSH (you need ssh-keygen installed).
- Supervisor is configured to automatically start the SSH server.

Parameters **client** – Connection to Docker (instance of `docker.Client`)

Returns The unique id of the base image.

`redock.base.`**download_image**(*client*, *repository*, *tag*)

Download the requested image. If the image is already available locally it won't be downloaded again.

Parameters

- **client** – Connection to Docker (instance of `docker.Client`)
- **repository** – The name of the image's repository.
- **tag** – The name of the image's tag.

`redock.base.`**find_base_image**(*client*)

Find the id of the base image that's used by Redock to create new containers. If the image doesn't exist yet it will be created using *create_base_image()*.

Parameters **client** – Connection to Docker (instance of `docker.Client`)

Returns The unique id of the base image.

`redock.base.`**find_named_image**(*client*, *repository*, *tag*)

Find the most recent Docker image with the given repository and tag.

Parameters

- **repository** – The name of the image's repository.
- **tag** – The name of the image's tag.

Returns The unique id of the most recent image available, or `None` if no images were matched.

Bootstrap configuration management system

Bootstrap is a minimal configuration management system. Right now it's just a toy module that I may or may not use to extend Redock beyond the existing `redock start`, `redock commit` and `redock kill` functionality and commands. Here is the design rationale behind Bootstrap (in its current form):

Specialized towards Debian Bootstrap is specialized towards Debian Linux (and its derivatives) because I have several years of hands on experience with Debian and Ubuntu Linux and because Docker currently gravitates to Ubuntu Linux (although this will probably change over time).

Based on SSH connections SSH is used to connect to remote hosts because it's the lowest common denominator that works with Docker, VirtualBox, XenServer and physical servers while being secure and easy to use.

Remote code execution using Python The execnet package is used to execute Python code on remote systems because I prefer the structure of Python code over shell scripts (Python avoids quoting hell). When Bootstrap connects to a remote system it automatically installs the system package `python2.7` on the remote system

because this is required to run execnet (on the other hand, execnet itself does not have to be installed on the remote system).

class redock.bootstrap.**Bootstrap**(*ssh_alias*)
The Bootstrap configuration management system is implemented as the class *Bootstrap*.

__init__(*ssh_alias*)
Initialize the configuration management system by creating an execnet gateway over an SSH connection. First we make sure the `python2.7` package is installed; without it execnet won't work.

Parameters **ssh_alias** – Alias of remote host in SSH client configuration.

execute(**command*, ***kw*)
Execute a remote command over SSH so that the output of the remote command (the standard output and standard error streams) is immediately visible on the local terminal. If no standard input is given, this allocates a pseudo-tty (using `ssh -t`) which means the operator can interact with the remote system should it prompt for input.

Raises *ExternalCommandFailed* if the remote command ends with a nonzero exit code.

Parameters

- **command** – A list with the remote command and its arguments.
- **input** – The standard input for the command (expected to be a string). This is an optional keyword argument. If this argument is given, no pseudo-tty will be allocated.

install_packages(**packages*)
Install the given system packages on the remote system.

Parameters **packages** – The names of one or more packages to install (strings).

rsync(*local_directory*, *remote_directory*, *cvs_exclude=True*, *delete=True*)
Copy a directory on the host to the container using rsync over SSH.

Raises *ExternalCommandFailed* if the remote command ends with a nonzero exit code.

Parameters

- **local_directory** – The pathname of the source directory on the host.
- **remote_directory** – The pathname of the target directory in the container.
- **cvs_exclude** – Exclude version control files (enabled by default).
- **delete** – Delete remote files that don't exist locally (enabled by default).

update_system_packages()
Perform a full upgrade of all system packages on the remote system.

upload_file(*pathname*, *contents*)
Create a file on the remote file system.

Parameters

- **pathname** – The absolute pathname on the remote system.
- **contents** – The contents of the file (a string).

exception redock.bootstrap.**ExternalCommandFailed**
Raised by *Bootstrap.execute()* and *Bootstrap.rsync()* when an external command fails (ends with a nonzero exit status).

Miscellaneous utility functions

class redock.utils.**Config**

Config encapsulates the bits of runtime configuration that Redock needs to persist to disk (to share state in between runs of Redock). UNIX file locking is used to guarantee that the datafile is not written to simultaneously by multiple processes (that could corrupt the state).

To use this class to update the configuration, use it like a context manager, like this:

```
>>> config = Config()
>>> with config as state:
...   state['containers'].clear()
```

When used like this, `state` is a dictionary which is saved to disk when the `with` block ends without raising an exception.

__init__()

load(*exists=True*)

Load the runtime configuration from disk. If the file doesn't exist yet an empty configuration is returned. The configuration contains a version number which enables graceful upgrades to the format.

Returns A dictionary with runtime configuration data.

class redock.utils.**RemoteTerminal**(*container_id*)

Attach to a running Docker container and show the output of the command(s) inside the container on the host's terminal. Can be used as a context manager or by manually calling *RemoteTerminal.attach()* and *RemoteTerminal.detach()*.

__init__(*container_id*)

Initialize the context manager for the `docker attach` process.

Parameters **container_id** – The id of the container to attach to (a string).

attach()

Start the `docker attach` subprocess.

detach()

Kill the `docker attach` subprocess.

redock.utils.**apt_get_install**(**packages*)

Generate a command to install the given packages with `apt-get`.

Parameters **packages** – The names of the package(s) to be installed.

Returns The `ap-get` command line as a single string.

redock.utils.**create_configuration_directory**()

Make sure Redock's local configuration directory exists.

redock.utils.**find_local_ip_addresses**()

To connect to a running Docker container over TCP we need to connect to a specific port number on an IP address associated with a local network interface on the host system (specifically *not* a loop back interface).

Returns A `set` of IP addresses associated with local network interfaces.

redock.utils.**generate_ssh_key_pair**()

Generate an SSH key pair for communication between the host system and containers created with Redock. Requires the `ssh-keygen` program.

redock.utils.**get_ssh_public_key**()

Get the contents of the SSH public key generated by Redock for use inside containers. If the SSH key pair hasn't been generated yet, it will be generated using *generate_ssh_key_pair()*.

Returns The contents of the `id_rsa.pub` file.

`redock.utils.`**`quote_command_line`**(*command*)
Quote the tokens in a shell command line.

Parameters **`command`** – A list with the command name and arguments.

Returns The command line as a single string.

`redock.utils.`**`select_ubuntu_mirror`**(*force=False*)
Find an Ubuntu mirror that is geographically close to the current location for use inside Docker containers. We remember the choice in a file on the host system so that we always configure the same mirror in Docker containers (if you change the mirror, `apt-get` has to download all package metadata again, wasting a lot of time).

`redock.utils.`**`slug`**(*text*)
Convert text to a "slug". Used by *redock.api.Container.ssh_alias*.

Parameters **`text`** – The original text, e.g. "Some Random Text!".

Returns The slug text, e.g. "some-random-text".

`redock.utils.`**`summarize_id`**(*id*)
Docker uses hexadecimal strings of 65 characters to uniquely identify containers, images and other objects. Docker's API almost always reports full IDs of 65 characters, but the `docker` program abbreviates these IDs to 12 characters in the user interface. We do the same because it makes the output more user friendly.

Parameters **`id`** – A hexadecimal ID of 65 characters.

Returns A summarized ID of 12 characters.

Python Module Index

r

Index

Symbols

A

B

C

D

E

F

G

I

K

L

N

Q

R

S

U