Docker-Fabric Documentation

Release 0.5.0

Matthias Erll

Dec 27, 2017

Contents

Docker-Fabric provides a set of utilities for controlling Docker on a local test machine or a remote production environment. It combines Fabric with extensions to docker-py in Docker-Map.

The project is hosted on GitHub.

CHAPTER 1

Features

- Integration of Docker-Map's container structure into Fabric deployments.
- Complements Docker API commands with command line shortcuts.
- Uses Fabric's SSH tunnel for connecting to the Docker Remote API.
- Fabric-like console feedback for Docker image and container management.

CHAPTER 2

Contents

2.1 Installation and configuration

2.1.1 Installation

The current stable release, published on PyPI, can be installed using the following command:

```
pip install docker-fabric
```

For importing YAML configurations for Docker-Map, you can install Docker-Fabric using

```
pip install docker-fabric[yaml]
```

Dependencies

The following libraries will be automatically installed from PyPI:

- Fabric (tested with >=1.8.0)
- docker-py (>=1.9.0)
- docker-map (>=0.8.0)
- Optional: PyYAML (tested with 3.11) for YAML configuration import

Docker service

Docker needs to be installed on the target machine. There used to be a utility task for this, but the required steps for installation tended to change too much too quickly for maintaining them properly. Please follow the Docker installation instructions according to your operating system.

Socat

The tool Socat is needed in order to tunnel local TCP-IP connections to a unix socket on the target machine. The `socat` binary needs to be in the search path. It is included in most common Linux distributions, e.g. for CentOS you can install it using `yum install socat`; or you can download the source code and compile it yourself.

2.1.2 Configuration

Docker service

On every target machine, Docker-Fabric needs access to the Docker Remote API and (optionally) to the command line client. With the default Docker configuration, this requires for the connecting SSH user to be in the *docker* user group. The group assignment provides access to the unix socket.

For assigning an existing user to that group, run

```
usermod -aG docker <user name>
```

Note that if you run this command with the same user (using *sudo*), you need to re-connect. Use `disconnect_all()` if necessary.

Tasks

If you plan to use the built-in tasks, include the module in your fabfile module (e.g. *fabfile.py*). Most likely you might want to assign an alias for the task namespace:

```
from dockerfabric import tasks as docker
```

Environment

In order to customize the general behavior of the client, the following variables can be set in Fabric's env. All of them are generally optional, but some are needed when tunnelling connections over SSH:

- `docker_base_url`: The URL of the Docker service. If not set, defaults to a socket connection in `/var/run/docker.sock`, which is also the default behavior of the *docker-py* client. If `docker_tunnel_remote_port` and/or `docker_tunnel_local_port` is set, the connection will be tunnelled through SSH, otherwise the value is simply passed to *docker-py*. For socket connections (i.e. this is blank, starts with a forward slash, or is prefixed with `http+unix:`, `unix:`), **socat** will be used to forward the TCP-IP tunnel to the socket.
- `docker_tunnel_local_port`: Set this, if you need a tunneled socket connection. Alternatively, the value `docker_tunnel_remote_port` is used (unless empty as well). This is the first local port for tunnelling connections to a Docker service on the remote. Since during simultaneous connections, a separate local port has to be available for each, the port number is increased by one on every new connection. This means for example, that when setting this to 2224 and connecting to 10 servers, ports from 2224 through 2233 will be temporarily occupied.
- `docker_tunnel_remote_port`: Port of the Docker service.
  - On TCP connections, this is the remote endpoint of the tunnel. If a different port is included in `docker_base_url`, this setting is ignored.
  - For socket connections, this is the initial local tunnel port. If specified by `docker_tunnel_local_port`, this setting has no effect.

- `docker_timeout`: Request timeout of the Docker service; by default uses `DEFAULT_TIMEOUT_SECONDS`.
- `docker_api_version`: API version used to communicate with the Docker service, as a string, such as `1.16`. Must be lower or equal to the accepted version. By default uses `DEFAULT_DOCKER_API_VERSION`.

Additionally, the following variables are specific for Docker registry access. They can be overridden in the relevant commands (`login()`, `push()`, and `pull()`).

- `docker_registry_user`: User name to use when authenticating against a Docker registry.
- `docker_registry_password`: Password to use when authenticating against a Docker registry.
- `docker_registry_mail`: E-Mail to use when authenticating against a Docker registry.
- `docker_registry_repository`: Optional; the registry to connect to. This will be expanded to a URL automatically. If not set, registry operations will run on the public Docker index.
- `docker_registry_insecure`: Whether to set the *insecure* flag on Docker registry operations, e.g. when accessing your self-hosted registry over plain HTTP. Default is `False`.

Examples

For connecting to a remote Docker instance over a socket, install **socat** on the remote, and put the following in your `fabfile`:

```
from fabric.api import env
from dockerfabric import tasks as docker

env.docker_tunnel_local_port = 22024  # or any other available port above 1024 of␣
↪your choice
```

If the remote Docker instance accepts connections on port 8000 from localhost (not recommended), use the following:

```
from fabric.api import env
from dockerfabric import tasks as docker

env.docker_base_url = 'tcp://127.0.0.1:8000'
env.docker_tunnel_local_port = 22024  # or any other available port above 1024 of␣
↪your choice
```

2.1.3 Checking the setup

For checking if everything is set up properly, you can run the included task *version*. For example, running

```
fab docker.version
```

against a local Vagrant machine (using the default setup, only allowing socket connections) and tunnelling through port 2224 should show a similar result:

```
[127.0.0.1] Executing task 'docker.version'
[127.0.0.1]
KernelVersion: 3.13.0-34-generic
Arch:          amd64
ApiVersion:    1.14
Version:       1.2.0
GitCommit:     fa7b24f
Os:            linux
```

```
GoVersion:      go1.3.1

Done.
Disconnecting from 127.0.0.1:2222... done.
```

2.2 Getting started

In order to connect with the Docker service, make sure that

1. Docker is installed on the remote machine;
2. **socat** is installed and in the remote's search path, if you are using the SSH tunnel;
3. and the SSH user has access to the service.

(For details, refer to *Installation and configuration*).

Calls to the Remote API can be made by using `docker_fabric()`. This function uses Fabric's usual SSH connection (creates a new one if necessary) and opens a separate channel for forwarding requests to the Docker Remote API.

Since this is merely a wrapper, all commands to *docker-py* are supported. Some additional functionality is provided by Docker-Map. However, instead of repeatedly passing in similar parameters (e.g. the service URL), settings can be preset globally for the project. Additionally, it provides a caching functionality for open tunnels and connections, which speeds up access to Docker significantly.

Short examples:

```
from dockerfabric.apiclient import docker_fabric
docker_fabric().version()
```

returns version information from the installed Docker service. This function is directly passed through to `docker-py` and formatted. The utility function:

```
docker_fabric().cleanup_containers()
```

removes all containers on the target Docker service that have exited.

For building images, use Docker-Map's `DockerFile` for generating an environment, and run:

```
docker_fabric().build_from_file(dockerfile, 'new_image_tag:1.0', rm=True)
```

2.3 Docker Remote API client for Fabric

`DockerFabricClient` is a client for access to a remote Docker host, which has been enhanced with some additional functionality. It is a Fabric adaption to `DockerClientWrapper` from the Docker-Map package, which again is based on docker-py, the reference Python client for the Docker Remote API.

`DockerClientWrapper` wraps some functions of *docker-py*, but most methods of its original implementation can also be used directly. This is described in more depth in the Docker-Map documentation. The following sections focus on the details specific for Docker-Fabric.

2.3.1 Basic usage

The constructor of `DockerFabricClient` accepts the same arguments as the *docker-py* implementation (`base_url`, `version`, and `timeout`), which are passed through. Moreover, `tunnel_remote_port` and `tunnel_local_port` are available. The following arguments of `DockerFabricClient` fall back to Fabric `env` variables, if not specified explicitly:

- `base_url`: `env.docker_base_url`
- `version`: `env.docker_api_version`
- `timeout`: `env.docker_timeout`
- `tunnel_remote_port`: `env.docker_tunnel_remote_port`
- `tunnel_local_port`: `env.docker_tunnel_local_port`

Although instances of `DockerFabricClient` can be created directly, it is more practical to do so implicitly by calling `docker_fabric()` instead:

- If parameters are set up in the Fabric environment, as listed in the *Environment* section, no further configuration is necessary.
- More importantly, existing client connections (and possibly tunnels) are cached and reused, similar to Fabric's connection caching. Therefore, you do not need to keep global references to the client around.

For example, consider the following task:

```
from dockerfabric.api import docker_fabric

@task
def sample_task():
    images = docker_fabric().images()
    ...
    containers = docker_fabric().containers(all=True)
    ...
```

The fist call to `docker_fabric()` opens the connection, and although you may choose to reference the client object with an extra variable, it will not use significantly more time to run `docker_fabric()` a second time. This becomes important especially on tunnelled connections.

New connections are opened for each combination of Fabric's host string and the Docker base URL. Therefore, you can run the task on multiple machines at once, just as any other Fabric task.

2.3.2 Working with multiple clients

Whereas `docker_fabric()` always opens the connection on the current host (determined by `env.host_string`), it may be beneficial to run Docker commands without a `host_string` or `roles` assignment if

- the set of clients, that are supposed to run container configurations, does not match the role definitions in Fabric;
- you do not feel like creating a separate task with host or role lists for each container configuration to be launched;
- or the client in some way require different instantiation parameters (e.g. different service URL, tunnel ports, or individual timeout settings).

Docker-Fabric enhances the client configuration from Docker-Map in `DockerClientConfiguration`. Setting any of `base_url`, `version`, `timeout`, `tunnel_remote_port` or `tunnel_local_port` overrides the global settings from the `env` variables mentioned in the last section. The object is mapped to Fabric's host configurations by the `fabric_host` variable.

If stored as a dictionary in `env.docker_clients`, configurations are used automatically by `container_fabric()`.

2.3.3 SSH Tunnelling

Docker is by default configured to only accept connections on a Unix socket. This is good practice for security reasons, as the socket can be protected with file system permissions, whereas the attack surface with TCP-IP would be larger. However, it also makes outside access for administrative purposes more difficult.

Fabric's SSH connection can tunnel connections from the local client to the remote host. If the service is only exposed over a Unix domain socket, the client additionally launches a **socat** process on the remote end for forwarding traffic between the remote tunnel endpoint and that Unix socket. That way, no permanent reconfiguration of Docker is necessary.

Tunnel functionality

Without a host connection in Fabric, the client attempts to make all connection locally (i.e. acts just like the *docker-py* client). With a `host_string` set, the `DockerFabricClient` opens a tunnel to forward traffic between the local machine and the Docker service on the remote host. Practically, a modified URL `tcp://127.0.0.1:<local port>` is passed to *docker-py*, where `<local port>` is the specified via `tunnel_local_port`. There are two tunnel methods, depending on the connection type to Docker:

1. If `base_url` indicates a Unix domain socket, i.e. it is prefixed with any `http+unix:`, `unix:`, `/`, or if it is left empty, **socat** is started on the remote end and forwards traffic between the remote tunnel endpoint and the socket.
2. In other cases of `base_url`, the client attempts to connect directly through the established tunnel to the Docker service on the remote end. The service has to be exposed to the port included in the `base_url` or set in `tunnel_remote_port` or.

As there needs to be a separate local port for every connection, the exact port `tunnel_local_port` is only used once between multiple clients. `DockerFabricClient` increases this by one for each additional host. From version 0.1.4, this also works with parallel tasks in Fabric.

Socat options

From version 0.2.0, **socat** does not expose a port on the remote end and therefore does not require further configuration. For information purposes, the client can however be set to echo the command to *stdout* by setting `env.socat_quiet` to `False`.

The utility task `reset_socat` removes **socat** processes, in case of occasional re-connection issues. Since **socat** no longer forks on accepting a connection, this should no longer occur.

2.3.4 Configuration example

Single-client configuration

Consider the following lines in your project's `fabfile.py`:

```
env.docker_tunnel_local_port = 2224
env.docker_timeout = 20
```

With this configuration, `docker_fabric()` in a task running on each host

1. opens a channel on the existing SSH connection and launches **socat** on the remote, forwarding traffic between the remote *stdout* and `/var/run/docker.sock` (the default base URL);
2. opens a tunnel through the existing SSH connection on port 2224 (increased by 1 for every additional host);
3. cancels operations that take longer than 20 seconds.

Multi-client configuration

In addition to the previous example, adding the following additional lines in your project's `fabfile.py`:

```
env.docker_clients = {
    'client1': DockerClientConfiguration(
        fabric_host='host1',
        timeout=40,  # Host needs longer timeouts than usual.
    ),
    'client2': DockerClientConfiguration(
        fabric_host='host2',
        interfaces={
            'private': '10.x.x.11',  # Host will be publishing some ports.
            'public': '178.x.x.11',
        },
    ),
}
```

A single client can be instantiated using:

```
env.docker_clients['client1'].get_client()
```

Similar to `docker_fabric()` each client per host and service URL is only instantiated once.

2.3.5 Registry connections

Docker-Fabric offers the following additional options for configuring registry access from the Docker host to a registry, as described in the *Environment* section. Those can be either set with keyword arguments at run-time, or with the environment variables:

- `username`: `env.docker_registry_user`
- `password`: `env.docker_registry_password`
- `email`: `env.docker_registry_mail`
- `registry`: `env.docker_registry_repository`
- `insecure_registry`: `env.docker_registry_insecure`

Whereas `env.docker_registry_insecure` applies to `login()`, `pull()`, and `push()`, the others are only evaluated during `login()`.

Note: Before a registry action, the local Docker client uses the *ping* endpoint of the registry to check on the connection. This has implications for using HTTPS connections between your Docker host(s) and the registry: Although everything is working fine on the Docker command line of the host, your client may reject the certificate because it does not trust it. This is very common with self-signed certificates, but can happen even with purchased ones. This behavior is defined by *docker-py*.

There are two methods to circumvent this issue: Either set `insecure_registry` (or `env.docker_registry_insecure`) to `True`; or add the certificate authority that signed the registry's certificate to your local trust store.

2.3.6 Docker-Map utilities

As it is based on Docker-Map, Docker-Fabric has also inherited all of its functionality. Regarding container maps, a few adaptions are described in the section *Fabric with Container maps*. The process of generating a *Dockerfile* and building an image from that is however very similar to the description in the Docker-Map documentation:

```
from dockermap.api import DockerFile

dockerfile = DockerFile('ubuntu', maintainer='ME, me@example.com')
...
docker_fabric().build_from_file(dockerfile, 'new_image')
```

2.4 Remote CLI client

Following a feature request on GitHub, an alternative client implementation has been added to Docker-Fabric and Docker-Map, for using a command-line-based interface to Docker. It supports the same options, methods, and arguments. However, it directly runs commands through Fabric instead of opening its an additional SSH channel. Due to different requirements in parsing output and handling errors this should be considered experimental.

Usage is very similar to the API client. There are two ways of changing between the two implementations:

1. By setting a Fabric environment variable:

   ```
   from fabric.api import env
   from dockerfabric.api import docker_fabric, container_fabric, CLIENT_CLI

   env.docker_fabric_implementation = CLIENT_CLI  # Default is CLIENT_API.
   ```

   This is the preferred method, as this also applies to utility tasks defined in Docker-Fabric.

2. Directly by importing a different method:

   ```
   from dockerfabric.cli import docker_cli, container_cli

   docker_cli().images()  # Instead of docker_fabric().images()
   container_cli().update(config_name)  # Instead of container_fabric().
   ↪update(config_name)
   ```

2.5 Miscellaneous utilities

Docker-Fabric supports deployments with a set of additional tools. Some components rely on the command line interface (CLI) of Docker, that can be run through Fabric.

2.5.1 Docker command line interface

Some functionality has been implemented using the Docker CLI mainly for performance reasons. It has turned out in earlier tests that the download speed of container and image data through the SSH tunnel was extremely slow.

This may be due to the tunnelling. The effect is further increased by the fact that the remote API currently does not compress any downstream data (e.g. container and image transfers to a client), although it accepts gzip-compressed upstream.

Containers

The two functions `copy_resource()` and `copy_resources()`, as the name may suggest, extract files and directories from a container. They behave slightly different from one another however: Whereas the former is more simple, the latter aims for flexibility.

For downloading some files and packaging them into a tarball (similar to what the `copy` function of the API would do) `copy_resource()` is more appropriate. Example:

```
from dockerfabric import cli
cli.copy_resource('app_container', '/var/log/app', 'app_logs.tar.gz')
```

This downloads all files from `/var/log/app` in the container `app_container` onto the host, packages them into a compressed tarball, and downloads that to your client. Finally, it removes the downloaded source files from the host.

If the copied resource is a directory, contents of this directory are packaged into the top level of the archive. This behavior can be changed (i.e. having the directory on the root level) by setting the optional keyword argument `contents_only=False`.

The more advanced `copy_resources()` is suitable for complex tasks. It does not create a tarball and does not download to your client, but can copy multiple resources, and modify file ownership (*chown*) as well as file permissions (*chmod*) after the download:

```
...
resources = ['/var/lib/app/data1', '/var/lib/app/data2']
temp_mapping = {'/var/lib/app/data1': 'd1', '/var/lib/app/data1': 'd2'}
cli.copy_resources('app_container', resources, '/home/data', dst_directories=temp_
↪mapping, apply_chmod='0750')
```

This example downloads two directories from `app_container`, stores them in a folder `/home/data`, and changes the file system permissions to read-write for the owner, read-only for the group, and no access for anyone else.

The directories would by default be stored within their original structure, but in this example are renamed to `d1` and `d2`. This is also possible with files. In order to override the generic fallback mapping (e.g. to something else than the resource name), add a key `*` to the dictionary. That way contents of multiple directories can be merged into one.

In case you would rather compress and download these files and directories, instead use `isolate_and_get()`. Similar to `copy_resource` contents are only stored temporarily and downloaded as a single gzip-compressed tarball, but with the same options as `copy_resources()`:

```
cli.isolate_and_get('app_container', resources, 'container.tar.gz', dst_
↪directories=temp_mapping)
```

It results in tar archive with `d1` and `d2` as top-level elements.

Since Docker also supports creating images from tar files, `isolate_to_image()` can generate an image that contains only the selected resources. Instead of a target file or directory, specify an image name instead:

```
cli.isolate_to_image('app_container', resources, 'new_image', dst_directories=temp_
↪mapping)
```

Note that the image at that point still has no configuration. In order for being able to run it as a container, some executable file needs to be included.

Images

As an alternative to the remote API `save_image`, `save_image()` stores the contents of an entire image into a compressed tarball and downloads that. It takes two arguments, the image and the tarball:

```
cli.save_image('app_image', 'app_image.tar.gz')
```

The function `flatten_image()` works different from `save_image`: It downloads the contents of an image and stores them in a new one. This can reduce the size, but comes with a couple of limitations.

A template container has to be created from the image first, and started with a command that makes no further modifications. For Linux images including the core utilities, such a command is typically `/bin/true`; where applicable it should be changed using the keyword argument `no_op_cmd`:

```
cli.flatten_image('app_image', 'new_image', no_op_cmd='/true')
```

If the second argument is not provided, the original image is overwritten. Like `isolate_to_image`, the original configuration is not transferred to the new image.

2.5.2 Fabric context managers

The following context managers complement `fabric.context_managers`. They are referenced in other areas of Docker-Fabric, and can also be used directly for deployments.

Note: Docker-Fabric includes more utility functions. Not all are described here, but are documented with the package `dockerfabric.utils`.

For some purposes it may be useful to create a temporary container from an image, copy some data from it, and destroy it afterwards. This is provided by `temp_container()`:

```
from dockerfabric.utils.containers import temp_containers

with temp_container('app_image', no_op_cmd='/true'):
    ...
```

In fact it is not a requirement that the command provided in the keyword argument `no_op_cmd` actually performs no changes. The command should finish without any interaction however, as the function waits before processing further commands inside that block. Further supported arguments are `create_kwargs` and `start_kwargs`, for cases where it is necessary to modify the create and start options of a temporary container.

Management of local files, e.g. for copying around container contents, is supported with two more temporary contexts: `temp_dir()` creates a temporary directory on the remote host, that is removed after leaving the context block. An alias should be assigned for use inside the context block:

```
from dockerfabric.utils.files import temp_dir

with temp_dir() as remote_tmp:
    cli.copy_resources('app_container', resources, remote_tmp, dst_directories=temp_
↪mapping, apply_chmod='0750')
    ...
...
# Directory is removed at this point
```

The local counterpart is `local_temp_dir()`: It creates a temporary folder on the client side:

```
from dockerfabric.utils.files import local_temp_dir

with local_temp_dir() as local_tmp:
    cli.copy_resource('app_container', '/var/log/app', os.path.join(local_tmp, 'app_
↪logs.tar.gz'))
    ...
```

2.6 Utility tasks for Fabric

Included utility tasks perform some basic actions within Docker. When importing them into your `fabfile.py`, you might want to assign an alias to the module, for having a clear task namespace:

```
import dockerfabric.tasks as docker
```

Then the following commands work directly from the command line, e.g. `fab <task name>`. A basic description of each task is displayed when running `fab --list` – the following sections describe a few further details.

2.6.1 General purpose tasks

Socat does not terminate after all connections to the host have been closed. Although this can be changed by setting `env.socat_fork` to `False`, there may be instances where it may be necessary to close the process manually, e.g. when the `fork` setting has just recently been set. The task `reset_socat()` finds **socat**'s process id(s) and sends a *kill* signal.

For configuration between containers and firewalls, the host's IP address can be obtained using the tasks `get_ip()` and `get_ipv6()`. Without further arguments it returns the address of the *docker0* interface. Specifying a different interface is possible via the first argument:

```
fab get_ip:eth0
```

returns the IPv4 address of the first network adapter. IPv6 addresses can additionally be expanded, e.g.

```
fab get_ipv6:eth0:True
```

returns the full address instead of the abbreviated version provided by `ifconfig`.

Tip: If you would like to handle this information directly in code, use the utility functions `get_ip4_address()` and `get_ip6_address()` instead.

2.6.2 Docker tasks

The following tasks are directly related to Docker and processed by the service on the remote host.

Information tasks

As mentioned in the *Installation and configuration* section, `version()` provides a similar output to running `docker version` on the command line.

Similarly, `list_images()` and `list_containers()` print a list of available images and running containers. The output is slightly different from the corresponding command line's. For `list_containers`

- Ports and multiple container names (e.g. linking aliases) are broken into multiple lines,
- images are by default shown without their registry prefix (can be changed by passing `short_image=False`),
- the absolute creation timestamp is printed,
- and by default all containers are shown (can be changed by passing an empty string as the first argument).

In the output of `list_images`

- parent image ids are shown,
- and also here the absolute creation timestamp is printed.

Container tasks

As of version 0.3.0, container maps are recommended to be set in `env.docker_maps` (as list or single entry) and multiple clients to be configured in `env.docker_clients`. In that setup, the lifecycle of containers, including their dependencies, can be entirely managed from the command line without creating individual tasks for them. The module `actions` contains the following actions:

- `create()` - Creates a container and its dependencies.
- `start()` - Starts a container and its dependencies.
- `stop()` - Stops a container and its dependents.
- `remove()` - Removes a container and its dependents.
- `startup()` - Creates and starts a container and its dependencies.
- `shutdown()` - Stops and removes a container and its dependents.
- `update()` - Updates a container and its dependencies. Creates and starts containers as necessary.
- `kill()` - Sends a `SIGKILL` signal to a single container. A different signal can be sent by specifying in the keyword argument `signal`, e.g. `signal=SIGHUP`.
- `pull_images()` - Pulls the image of the specified container if it is absent, and all of its dependency container images.
- `script()` - Uploads and runs a script inside a container, which is created specifically for that purpose, along with its dependencies. The container is removed after the script has completed.
- `single_cmd()` - Similar to `script()`, but not uploading contents beforehand, for running a self-contained command (e.g. Django *migrate*, Redis *flusdhdb* etc.). If this produces files, the results can be downloaded however.

Note: There is also a generic action `perform()`. Performs an action on the given container map and configuration. There needs to be a matching implementation in the policy class.

Given the lines in `fabfile.py`:

```
from dockerfabric import yaml, actions

env.docker_maps = yaml.load_map_file('/path/to/example_map.yaml', 'example_map')
env.docker_clients = yaml.load_clients_file('/path/to/example_clients.yaml')
```

The web server from the *YAML import* example may be started with

```
fab actions.startup:example_map,web_server
```

runs the web server and its dependencies. The command

```
fab actions.update:example_map,web_server
```

stops, removes, re-creates, and starts the container if the image as specified in the container configuration (e.g. `nginx:latest`) has been updated, or mapped volumes virtual filesystems are found to mismatch the dependency containers' shared volumes.

Maintencance tasks

The maintenance tasks `cleanup_containers()`, `cleanup_images()`, and `remove_all_containers()` simply call the corresponding methods of `DockerFabricClient`:

- `cleanup_containers()` removes all containers that have the *Exited* status;
- `cleanup_images()` removes all untagged images, optionally with the argument `True` also images without a `latest` tag. Additional tags can be specified by setting the environment variable `docker_keep_tags`.
- `remove_all_containers()` stops and removes all containers from the host.

Image transfer

Especially during the initial deployment you may run into a situation where manual image transfer is necessary. For example, when you plan to use your own registry, but would like to use your own web server image for a reverse proxy, the following tasks help to download the image from your build system to the client, and upload it to the production server:

Use `save_image()` with two arguments: Image name or id, and file name. If the file name is omitted, the image is stored in the current working directory, as `<image>.tar.gz`. For performance reasons, `save_image()` currently relies on the command line client. The compressed tarball is generated on the host.

```
fab docker.save_image:new_image.tar.gz
```

In reverse, `load_image()` uploads a local image to the Docker host. In this case the Docker Remote API is used. It accepts plain and gzip-compressed tarballs. The local image file name is the first argument. Since the API often times out for larger images (default is 60 seconds), the period is extended temporarily to 120 seconds. This can optionally be adjusted with a second argument, e.g.

```
fab docker.load_image:new_image.tar.gz:600
```

for an image that might take longer to upload due to a slow connection.

2.7 Fabric with Container maps

Using `docker_fabric()`, the standard API to Docker is accessible in a similar way to other Fabric commands. With Docker-Map, the API has been enhanced with a set of utilities to configure containers and their dependencies.

The configuration is in full discussed in the Docker-Map configuration, along with examples. This section explains how to apply this to Docker-Fabric.

2.7.1 Managing containers

In order to have the map available in your Fabric project, it is practical to store a reference in the global `env` object. The example from Docker-Map could be initialized with reference to other configuration variables:

```
env.host_root_path = '/var/lib/site'
env.registry_prefix = 'registry.example.com'
env.nginx_config_path = 'config/nginx'
env.app1_config_path = 'config/app1'
env.app2_config_path = 'config/app2'
env.app1_data_path = 'data/app1'
env.app2_data_path = 'data/app2'

env.docker_maps = ContainerMap('example_map', {
    'repository': env.registry_prefix,
    'host_root': env.host_root_path,
    'web_server': { # Configure container creation and startup
        'image': 'nginx',
        'binds': {'/etc/nginx': ('env.nginx_config_path', 'ro')},
        'uses': 'app_server_socket',
        'attaches': 'web_log',
        'exposes': {
            80: 80,
            443: 443,
        },
    },
    'app_server': {
        'image': 'app',
        'instances': ('instance1', 'instance2'),
        'binds': (
            {'app_config': 'ro'},
            'app_data',
        ),
        'attaches': ('app_log', 'app_server_socket'),
        'user': 2000,
        'permissions': 'u=rwX,g=rX,o=',
    },
    'volumes': { # Configure volume paths inside containers
        'web_log': '/var/log/nginx',
        'app_server_socket': '/var/lib/app/socket',
        'app_config': '/var/lib/app/config',
        'app_log': '/var/lib/app/log',
        'app_data': '/var/lib/app/data',
    },
    'host': { # Configure volume paths on the Docker host
        'app_config': {
            'instance1': env.app1_config_path,
            'instance2': env.app2_config_path,
        },
        'app_data': {
            'instance1': env.app1_data_path,
            'instance2': env.app2_data_path,
        },
    },
})
```

In order to use this configuration set, create a `ContainerFabric` instance from this map. For example, in order to launch the web server and all dependencies, run:

```
from dockerfabric.api import container_fabric

container_fabric().startup('web_server')
```

`ContainerFabric` (aliased with `container_fabric()`) calls `docker_fabric()` with the host strings on demand, and therefore runs the selected map on each host where required.

`env.docker_maps` can store one container map, or a list / tuple of multiple container maps. You can also store host definitions in any variable you like and pass them to `container_fabric`:

```
container_fabric(env.container_maps)
```

Multi-client configurations are automatically considered when stored in `env.docker_clients`, but can also be passed through a variable:

```
container_fabric(maps=custom_maps, clients=custom_clients)
```

2.7.2 YAML import

Import of YAML files works identically to Docker-Map's implementation, but with one more added tag: `!env`. Where applied, the following string is substituted with the current value of a corresponding `env` variable.

When using the `!env` tag, the order of setting variables is relevant, since values are substituted at the time the YAML file is read. For cases where this is impractical some configuration elements support a 'lazy' behavior, i.e. they are not resolved to their actual values until the first attempt to access them. In order to use that, just apply `!env_lazy` in place of `!env`. For example volume paths and host ports can be assigned with this tag instead. A full list of variables supporting the late value resolution is maintained in the Docker-Map documentation.

Note: If the variable is still missing at the time it is needed, a `KeyError` exception is raised.

In order to make use of the `!env` and `!env_lazy` tag, import the module from Docker-Fabric instead of Docker-Map:

```
from dockerfabric import yaml

env.docker_maps = yaml.load_map_file('/path/to/example_map.yaml', 'example_map')
env.docker_clients = yaml.load_clients_file('/path/to/example_clients.yaml')
```

One more difference to the Docker-Map `yaml` module is that `load_clients_file()` creates object instances of `DockerClientConfiguration()`. The latter consider specific settings as the tunnel ports, which are not part of Docker-Map.

Container map

In the file `example_map.yaml`, the above-quoted map could be represented like this:

```
repository: !env registry_prefix
host_root: /var/lib/site
web_server:
  image: nginx
  binds:
    /etc/nginx:
    - !env nginx_config_path
```

```
    - ro
  uses: app_server_socket
  attaches: web_log
  exposes:
    80: 80
    443: 443
app_server:
  image: app
  instances:
  - instance1
  - instance2
  binds:
  - app_config: ro
  - app_data:
  attaches:
  - app_log
  - app_server_socket
  user: 2000
  permissions: u=rwX,g=rX,o=
volumes:
  web_log: /var/log/nginx
  app_server_socket: /var/lib/app/socket
  app_config: /var/lib/app/config
  app_log: /var/lib/app/log
  app_data: /var/lib/app/data
host:
  app_config:
    instance1: !env app1_config_path
    instance2: !env app2_config_path
  app_data:
    instance1: !env app1_data_path
    instance2: !env app2_data_path
```

Client configurations

With some modifications, this map could also run a setup on multiple hosts, for example one web server running as reverse proxy for multiple identical app servers:

```
env.docker_maps.update(
    web_server={
        'clients': 'web',
        'uses': [],  # No longer look for a socket
    },
    app_server={
        'clients': ('apps1', 'apps2', 'apps3'),
        'attaches': 'app_log',  # No longer create a socket
        'exposes': [(8443, 8443, 'private')],  # Expose a TCP port on 8443 of the␣
↪private network interface
    }
)
```

The modifications could of course have been included in the aforementioned map right away. Moreover, all of this has to be set up in the web server's and app servers' configuration accordingly.

A client configuration in `example_clients.yaml` could look like this:

```
web:
  fabric_host: web_host  # Set the Fabric host here.
apps1:
  fabric_host: app_host1
  interfaces:
    private: 10.x.x.21   # Provide the individual IP address for each host.
apps2:
  fabric_host: app_host2
  interfaces:
    private: 10.x.x.22
apps3:
  fabric_host: app_host3
  interfaces:
    private: 10.x.x.23
```

Since there is no dependency indicated by the configuration between the web and app servers, two startup commands are required; still they will connect to each host as necessary:

```
with container_fabric() as cf:
    cf.startup('web_server')
    cf.startup('app_server')
```

In addition to creating and starting the containers, ports will be bound to each private network adapter individually.

2.8 Change History

2.8.1 0.5.0

- Minor addition to CLI client.

2.8.2 0.5.0b4

- Fixed image id parsing after successful build.

2.8.3 0.5.0b3

- Added new CLI functions and utility task for volumes, as now implemented in Docker-Map 0.8.0b3.

2.8.4 0.5.0b2

- Fixed setup (requirements).

2.8.5 0.5.0b1

- Adapted to recent Docker-Map developments, which includes networking support and improved error handling.
- Dropped setup tasks for Docker and Socat. Socat is included in most Linux distributions, so that the task of compiling it from source was likely not used. Installation instructions for Docker have changed too frequently, and a correct (supported) setup depends strongly on the environment it is installed in, with more aspects than these simple tasks could consider.

2.8.6 0.4.2

- Added `top` method to CLI client.

2.8.7 0.4.1

- Fixed side-effects of modifying the `base_url` for SSH tunnels, causing problems when re-using a client returned by the `docker_fabric()` function (Issue #12).
- Added `version` method to CLI client.
- Added `env.docker_cli_debug` for echoing commands.
- API clients' `remove_all_containers` now forwards keyword arguments.

2.8.8 0.4.0

- Added Docker-Map's new features (keeping certain tags during cleanup and adding extra tags during build).
- Added experimental *Remote CLI client*. This has changed the module structure a bit, but previous imports should still work. From now on however, `docker_fabric` and `container_fabric` should be imported from `dockerfabric.api` instead of `dockerfabric.apiclient`.
- Fixed installation task for CentOS.

2.8.9 0.3.10

- Updated Docker service installation to follow reference instructions.
- Added separate utility tasks for CentOS.
- Fixed build failures in case of unicode errors.

2.8.10 0.3.9

- Client configuration is not required, if defaults are used.

2.8.11 0.3.8

- Implemented local (faster) method for adjusting permissions on containers.
- Fixed issues with non-existing directories when downloading resources from containers.

2.8.12 0.3.7

- Minor logging changes.
- Make it possible to set raise_on_error with pull (PR #3)

2.8.13 0.3.6

- Added script and single-command actions.

2.8.14 0.3.5

- docker-py update compatibility.

2.8.15 0.3.4

- Added Fabric error handling when build fails.
- Fixed re-use of existing local tunnels when connections are opened through different methods.

2.8.16 0.3.3

- More consistent arguments to connection behavior.

2.8.17 0.3.2

- Added `!env_lazy` YAML tag.
- Fixed bug on local connections.

2.8.18 0.3.1

- Added restart utility task.

2.8.19 0.3.0

- Added Docker-Map's new features (host-client-configuration and multiple maps).

2.8.20 0.2.0

- Revised SSH tunnelling of Docker service connections; not exposing a port on the host any longer.

2.8.21 0.1.4

- Intermediate step to 0.2.0, not published on PyPI.
- Better tolerance on missing parameters.
- Improved multiprocessing behavior (parallel tasks in Fabric).

2.8.22 0.1.3

- Only setup fix, no functional changes.

2.8.23 0.1.2

- Added more utility tasks, functions, and context managers.
- Improved output format of builtin tasks.
- Cleanups and fixes in utility functions.

2.8.24 0.1.1

- Added YAML import.
- Added default host root path and repository prefix.
- Added Docker registry actions.
- Added import/export utility functions.
- Attempts to fix reconnect and multiple connection issues.

2.8.25 0.1.0

Initial release.

CHAPTER 3

Status

Docker-Fabric is being used for small-scale deployment scenarios in test and production. It should currently considered beta, due to pending new features, generalizations, and unit tests.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search