Docker-Map Documentation

Release 1.0.0

Matthias Erll

Jan 15, 2018

Contents

Docker-Map enhances docker-py, the Docker Remote API client library for Python. It provides a set of utilities for building Docker images, create containers, connect dependent resources, and run them in development as well as production environments.

For connections through SSH tunnels, command-line tools, and additional deployment utilities, see Docker-Fabric.

The project is hosted on GitHub.

CHAPTER 1

Features

- Configuration of container landscapes, including dependencies between containers, networks, and volumes.
- Update check of containers, volumes, and networks against their running configuration. Consistency check of shared volume paths and re-creation of inconsistent or outdated containers.
- Utility client functions. Some of these (e.g. `cleanup_containers`) have also been implemented on later versions of Docker directly; however this library can still provide more fine-grained control.
- Creation of complex *Dockerfile*'s through Python code.
- Simplified transfer of build resources (Context) to the Remote API.

CHAPTER 2

Comparison to Compose

The functionality is quite similar to Docker Compose. When development of this library started, a predecessor of Compose was known as *Fig*. Docker-Map was mainly developed for the following reasons:

- *Docker-Map* is intended to provide an API that could be embedded into other available systems such as configuration management. This implies that configuration can be implemented through code (although YAML input is also available) and functions can be invoked through Python directly.
- *Docker-Map* is not a command-line utility, and it can operate on multiple clients. For example Docker-Fabric is one implementation that connect to other clients via SSH, but can also run on the same machine.
- *Fig* and also its successor *Compose* target development and staging environments. *Docker-Map* aims to handle staging to production environments by allowing for embedding external variables, e.g. from configuration management. Although it can be combined with other tools to be suitable for development and CI, *Compose* is more common to use and might be the first choice there.

CHAPTER 3

Contents

3.1 Installation and configuration

3.1.1 Installation

The first version of the Docker API Python client was released as `docker-py`. Version 2.x of the now so-called Docker SDK for Python introduced some breaking changes and removed backwards compatibility with older Docker Remote API versions. It was published with the `pip` package name `docker`. Since Docker-Map currently supports *both* versions which cannot be installed at the same time, none of them will be installed by default. You can either

- install one of `docker` or `docker-py`,
- or use one of the extra-options below.

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

```
pip install docker-map[docker]
```

This also installs the Docker SDK for Python 2.x. If you want to install the older implementation (version 1.x), use the following instead:

```
pip install docker-map[legacy]
```

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

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

Upgrading

If you were using an older version (< 1.0.0) of Docker-Map and want to migrate from `docker-py` (1.x) to the new `docker` (2.x) library, uninstall the older one first, and then reinstall:

```
pip uninstall docker-py
pip install docker
```

Docker service

Docker needs to be installed on the target machine, and needs to be accessible either through a Unix socket or TCP/IP. If you do not wish to expose a public network port from the host running Docker, consider using Docker-Fabric. Otherwise, for setting up a secure connection, refer to the Docker documentation.

3.1.2 Imports

The most essential classes are collected in the top-level `api` module for convenience. For example, you can use:

```python
from dockermap.api import ContainerMap
```

rather than importing the class from the actual source module such as:

```python
from dockermap.map.container import ContainerMap
```

3.2 Getting started

3.2.1 Simple Dockerfile

For deriving a new image from an existing *ubuntu* base image, you can use the following code:

```python
from dockermap.api import DockerClientWrapper, DockerFile

client = DockerClientWrapper('unix://var/run/docker.sock')
with DockerFile('ubuntu:latest') as df:
    df.run('apt-get update')
    df.run('apt-get -y upgrade')
    client.build_from_file(df, 'new_base_image', add_latest_tag=True, rm=True)
```

3.2.2 Docker image with files

For adding files during the build process, use the `add_file()` function. It inserts the *ADD* command into the Dockerfile, but also makes sure that file is part of the context tarball:

```python
from dockermap.api import DockerClientWrapper, DockerFile

client = DockerClientWrapper('unix://var/run/docker.sock')
with DockerFile('ubuntu:latest') as df:
    df.add_file('/home/user/myfiles', '/var/lib/myfiles')
    client.build_from_file(df, 'new_image')
```

3.2.3 Removing all containers

The `DockerClientWrapper` has enhances some of the default functionality of *docker-py* and adds utility functions. For example, you can remove all stopped containers from your development machine by running:

```
from dockermap.api import DockerClientWrapper

client = DockerClientWrapper('unix://var/run/docker.sock')
client.cleanup_containers()
```

3.2.4 Configuring containers

A `ContainerMap` provides a structure for mapping out container instances along with their dependencies.

A simple example could be a web server an an application server, where the web server uses Unix sockets for communicating with the application server. The map could look like this:

```
from dockermap.api import ContainerMap

container_map = ContainerMap('main', {
    'nginx': { # Configure container creation and startup
        'image': 'nginx',
        'binds': {'nginx_config': 'ro'},
        'uses': 'uwsgi_socket',
        'attaches': 'nginx_log',
        'exposes': {
            80: 80,
            443: 443,
        },
    },
    'uwsgi': {
        'binds': (
            {'uwsgi_config': 'ro'},
            {'app_config': 'ro'},
            'app_data',
        ),
        'attaches': ('uwsgi_log', 'app_log', 'uwsgi_socket'),
        'user': 2000,
        'permissions': 'u=rwX,g=rX,o=',
    },
    'volumes': { # Configure volume paths inside containers
        'nginx_config': '/etc/nginx',
        'nginx_log': '/var/log/nginx',
        'uwsgi_config': '/etc/uwsgi',
        'uwsgi_socket': '/var/lib/uwsgi/socket',
        'uwsgi_log': '/var/log/uwsgi',
        '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
        'nginx_config': '/var/lib/site/config/nginx',
        'uwsgi_config': '/var/lib/site/config/uwsgi',
        'app_config': '/var/lib/site/config/app',
        'app_data': '/var/lib/site/data/app',
    },
})
```

Note:

- By default an instantiation of such a map performs a brief integrity check, whether all aliases as used in container configurations have been defined in *host* and *volumes* assignments.
- *Attached* volumes are Docker containers based on a minimal launchable image, that are created for the sole purpose of sharing a volume. In this example, the *nginx* container will have access to *uwsgi_socket*, but none of the other shared volumes.
- The aforementioned *permissions* in the *uwsgi* container assume that the working user in the *nginx* container is part of a group with the id *2000*. If this is not the case, you have to open up *permissions*, e.g. to `u=rwX,g=rX,o=rX`.
- Although it is out of scope of this introduction, the recommended method for configuring container maps is the *import of YAML files*. It is syntactically simpler than Python code.

This map can be used with a `MappingDockerClient`:

```
from dockermap.api import DockerClientWrapper, MappingDockerClient

map_client = MappingDockerClient(container_map, DockerClientWrapper('unix://var/run/
↪docker.sock'))
map_client.startup('nginx')
```

This performs the following tasks:

- Resolve dependencies in order to determine which containers to start prior to *nginx*. In this case, *nginx* needs access to some *uwsgi_socket* volume. The latter is provided by starting *uwsgi*.
- Create containers for sharing attached volumes, and assign configured user (*chown*) and access permissions (*chmod*).
- Create and start containers *uwsgi* and *nginx* in that order, passing the necessary parameters to *docker-py*.

If images become updated on the Docker host, running containers can easily use the newer versions:

```
map_client.update('nginx')
```

Along the aforementioned dependency path, every container is stopped, removed, re-created and restarted as necessary if

- the image id does not match the current tag specification, e.g. since a new image version has been pulled,
- the container is stopped and its exit status indicates that it cannot be restarted,
- a linked container is missing,
- the virtual filesystems refer to the same path inside the container, but on the host they do not match (e.g. due to container updates along the dependency path),
- *port assignments* have changed, or
- `environment`, `command`, or `entrypoint` have been modified in the `create_options` since the current container was created.

Non-running containers are simply started during this process, if their configuration corresponds with their current state.

3.3 Building images with DockerFile

The functionality of `DockerFile` is centered around creating *Dockerfile*'s for Docker images. Although it is not particularly hard to write them directly, doing so requires you to remember what to configure where. In some instances (e.g. commands, ports etc.) this may be done at run-time using a configuration utility. However, if there are more dynamic elements, e.g. paths and version numbers, you can end up having to change them in multiple places.

This implementation aims to make Dockerfiles easy to generate by Python code. Approaches in detail may vary, i.e. some may prefer to insert commands one-by-one, whereas others would rather use a format string to insert variables. It is possible to combine such methods.

3.3.1 Basic instantiation

A new *Dockerfile* can be created with the following commands:

```
from dockermap.api import DockerFile

df = DockerFile('ubuntu', maintainer='ME, me@example.com', initial='RUN apt-get␣
↪update\nRUN apt-get -y upgrade')
```

The first argument is the base image, as every new Docker image should have one. The `maintainer` argument is optional, and is written with a `MAINTAINER` prefix. Afterwards, the contents of `initial` (also optional) are written to the Dockerfile.

Internally, instantiation creates a string buffer and some configuration variables. All action commands where the order is relevant, e.g. `RUN`, are written to this buffer immediately, whereas configuration commands such as `EXPOSE` are delayed until a finalization step.

Except for being embedded in a context tarball, the Dockerfile is never actually stored by itself. If you wish to do so, you can use `save()`.

3.3.2 Action commands

The following actions are performed on the string buffer immediately:

Initial contents

Passing in a base image, a maintainer, or `initial` contents on instantiation. Plain Dockerfile commands can be used in `initial`, as found in the Dockerfile reference. The base image is prefixed with `FROM`, whereas the maintainer is inserted with `MAINTAINER`. None of the `initial` commands are processed any further, so they should be formatted properly and contain line breaks.

Run commands

In order to insert `RUN` commands for execution during the build process, use `run()` and `run_all()`. They are convenience methods for `prefix('RUN', 'command')` and `prefix_all('RUN', 'command 1', 'command 2', ...)`.

Adding files

Files can be added using `add_file()` and `add_archive()`. The former adds a single file or directory, whereas the latter adds the contents of a tar archive. By default, all files and directories will be inserted at the root of the container's filesystem, maintaining their original structure of subdirectories where applicable.

Inserting a file or archive also adds an entry to a list which is used for building the *context* tarball. The latter carries all file-based information that is later sent to the Docker Remote API, including the finalized *Dockerfile*. Files and directories will simply be added to the context archive, whereas archives' contents are extracted and recompressed without storing additional temporary files.

Target directories inside the image can be specified for `add_file()` using further arguments. For archives, this is currently not supported, so existing tarballs should be structured in a way suitable for the image.

For example, a file may also be added with the following arguments:

```
dockerfile.add_file('~/my_file', '/new_dir/my_file', '/another_file', expanduser=True,
↪ remove_final=True)
```

Explanation:

- The first argument is the current actual place in the file system from where Docker-Map is run. If this includes variables, such as the current user home ~, `expanduser` should be set to `True` for resolving it to an absolute path name. Similarly, environment variables can be used when passing `expandvars=True`.
- The second argument defines the target path in the final image. By default, the file would have ended up in `/my_file`.
- The third argument is also optional, and specifies the path inside the context archive. By default it is identical to the image's destination path, and can be used in case conflicts arise from adding multiple files or directories with identical names.
- `remove_final` inserts a removal command (e.g. `RUN rm -Rf /new_dir/my_file`) at the end of the Dockerfile, but before configuration commands. You may want to set this to clean up the file system of the final image from files and directories that were only needed during the build process. Please note that due to the file system layering that Docker uses, this will not actually make the image smaller.

Comments and blank lines

Comments can be inserted with `comment()`, which is only a convenience for `prefix('#', 'comment')`. Passing `None` inserts an empty comment line. Blank lines are inserted with `blank()`. Note that these only have an effect if you actually store the Dockerfile somewhere.

Miscellaneous Docker commands

Any Dockerfile command, or a series thereof, can be inserted with `prefix()` and `prefix_all()`. These insert strings prefixed with a Dockerfile command. Following convenience methods should be preferred where available.

Direct write access

Strings with Dockerfile contents may also be written directly using `write()` and `writeline()` (same, but appends a line break) and `writelines()` (for multiple). They are not further processed besides that.

3.3.3 Configuration commands

The following are set as properties to a Dockerfile. They are appended as soon as `finalize()` is called. Afterwards no more changes are allowed to the object. Typically it is not necessary to call `finalize()` manually.

Volumes

Setting `volumes` defines the list of volumes that a container in its default configuration will share. The list will be inserted prefixed with a `VOLUME` command, before any other of the following finalizing commands.

Entry point and default command

`entrypoint` and `command` do the same as inserting `ENTRYPOINT` and `CMD` in the Dockerfile. They can be set either as a list/tuple of strings, or a single string separated with spaces. Depending on `command_shell`, they are either written as a shell command in the Dockerfile (i.e. with spaces) or as an exec command (i.e. as a list).

The `command_user` property sets the default user for `COMMAND` and `ENTRYPOINT`. It is therefore inserted directly before them. In contrast to inserting the `USER` command directly, this does not change the user for other commands in the Dockerfile. You can still use `prefix('USER', 'username')` if you need to change users during the build process.

Similarly, `command_workdir` sets the working directory for `ENTRYPOINT`, `CMD`. It does however not change directories immediately, i.e. does not affect `RUN` commands.

Exposed ports

`expose` can be set as a single string, integer, or as a list or tuple thereof. It will be written to the Dockerfile with the `EXPOSE` command; if applicable, multiple ports are separated with spaces.

3.3.4 Building the Docker image

For starting the build process, pass the `DockerFile` to the Docker Remote API with the enhanced client method `build_from_file()`:

```
from dockermap.api import DockerClientWrapper, DockerFile

client = DockerClientWrapper('unix://var/run/docker.sock')
dockerfile = DockerFile('ubuntu', maintainer='ME, me@example.com')
dockerfile.add_file(...)
dockerfile.run_all(...)
...
client.build_from_file(dockerfile, 'new_image')
```

3.4 Working with the DockerContext

The context is a tar file, that is submitted to the API in order to define the image building process. It has to include the Dockerfile and all necessary other files. The latter are all files referenced to in any `ADD` command. For syntax of `ADD` is:

```
ADD <source> <destination>
```

where source in this case refers to the path inside the build context, i.e. the tar file root.

When you add files to a DockerFile using add_file() and add_archive(), it generates a list of used files and directories. These can be automatically added to a DockerContext. For most common build scenarios, you may start the build process directly by calling build(), e.g.:

```
client.build_from_file(dockerfile, 'new_base_image', add_latest_tag=True, rm=True)
```

This automatically generates the context and uploads it. However, the context can also be modified further beforehand.

3.4.1 Creating a DockerContext

For generating a DockerContext explicitly from an existing DockerFile, just pass it to the constructor:

```
from dockermap.api import DockerContext

with DockerContext(dockerfile) as context:
    ...
```

This will create a new compressed tar archive, add the generated Dockerfile (string buffer) and the referenced files. Note that the DockerFile fill be finalized and cannot be modified further after this.

The with (Python context manager syntax) should be used, since DockerContext generates a temporary file which is automatically removed at the end of the block.

It is also possible to pass in a path to a file, e.g.:

```
from dockermap.api import DockerContext

with DockerContext(path_to_dockerfile) as context:
    ...
```

In that case, referenced files are not added automatically and have to be placed using the following methods.

3.4.2 Adding more files

DockerContext provides the methods add() and addfile(), which refer to tarfile.TarFile.add() and tarfile.TarFile.addfile(). Besides that, addarchive() copies the contents of another tar archive, including the structure of files and directories.

For using addfile(), a tarfile.TarInfo object is required. You can obtain that using gettarinfo(), which calls tarfile.TarFile.gettarinfo().

3.4.3 Using the context

Before sending the file to the Docker Remote API, the underlying tar archive has to be closed. This is handled by finalize(). Note that the underlying tar archive is closed from that point and can no longer be modified.

The context tarball is transferred to Docker with build_from_context():

```
from dockermap.api import DockerClientWrapper, DockerContext

client = DockerClientWrapper('unix://var/run/docker.sock')
with DockerContext(path_to_dockerfile) as context:
    ...
    context.finalize()
    client.build_from_context(context, 'new_image')
```

In fact, `dockermap.map.base.DockerClientWrapper.build_from_file()` is only a convenience wrapper around it. It finalizes the `DockerContext` object automatically.

3.4.4 Getting more information

Although it may not be relevant in practice, the entire context tarball could be stored to an archive using `save()`. By default this is a gzip compressed tar archive, but the actual method (which also needs to be specified to the Docker Remote API) can be read from the `stream_encoding` attribute:

- `gzip` means that the tarball is in the default format, i.e. *.tar.gz*;
- `bzip2` indicates a bzip compressed tar archive;
- and `None` means that the tar archive is not compressed.

In case you would like to know the name of the temporary underlying tar archive, without making a copy through `save()`, the property `name` is available.

3.5 Shortcuts for Dockerfiles

A couple of common commands in a *Dockerfile* require writing a lot of repetitive code. In combination with variables from Python code, additional adaptions (e.g. escaping of strings, reformatting of certain parameters) has to be made. The `shortcuts` module includes some utilities – more precisely string formatters – for use in a *Dockerfile*. They are also included in other modules of Docker-Map. Of course the generated commands can also be applied to any other command line setting, e.g. `run` calls in *Fabric*.

3.5.1 Users and groups

Since Docker does not use the host system's names, you either have to rely on only user ids, or create users and groups within the image. The former may not always be sufficient. Therefore, it is more practical to include commands such as `RUN adduser username ...`. Additionally, when sharing volumes between images, user and group ids should be consistent between the containers that are accessing them.

The utility `adduser()` generates a *adduser* command with the arguments `username` and `uid` – other keyword arguments are optional. The optional default values assume that typically, you need a user for running programs, but not for login. They can be overwritten in any other case:

- `system`: Create a system user; default is `False`.
- `no_login`: Do not allow the user to login and skip creation of a *home* directory; default is `True`.
- `no_password`: Do not issue a password for the user. It is set to `False` by default, but implied by `no_login`.
- `group`: Add a user group for the user. It is set to `False` by default. In Dockerfiles, you might want to call `addgroupuser()` instead for making the id predicable.
- `gecos`: Optional, but should usually be set to appropriate user information (e.g. full name) when `no_login` is set to `False`, as it avoids interactive prompts.

Similarly, `addgroup()` creates a *addgroup* command with the arguments `groupname` and `gid`. The optional `system` keyword argument decides whether to create a system user. For adding users to a group, use `assignuser()` with the arguments `username` and a list of groups in `groupnames`.

The three aforementioned functions can be comined easily with `addgroupuser()`. Like the `adduser()` shortcut, it has two mandatory arguments `username` and `uid`, and provides the keyword arguments `system`, `no_login`, `no_password`, and `gecos` with identical defaults. Additionally, a list of groups can be passed in `groupnames`. A user and group are created with identical name and id. When needed, the user is additionally added to a list of additional groups.

For example, for creating a user `nginx` for running web server workers with, inlcude the following commands:

```
df = DockerFile(...)
... #  Additional build commands
df.run(addgroupuser('nginx', 2000))
```

If you are sharing files or a socket with an app server container named *apps* and the group id *2001*, the following code creates that group and assigns the web server user to it:

```
df = DockerFile(...)
... #  Additional build commands
df.run_all(
    addgroup('apps', 2001),
    addgroupuser('nginx', 2000, ['apps']),
)
```

Tip: The user and group names, as well as their ids, are only written here as literals for illustration purposes. The main intention of the `DockerFile` implementation is that you do not hard-code these settings, but instead refer to variables.

Note: *adduser* and *addgroup* are specific to Debian-based Linux distributions. Therefore, they will be replaced with more system-independent commands in future versions.

User names of most Docker-Map functions are formatted by `get_user_group()`. It accepts the following input, which is returned as a string in the format `user:group`:

- Tuple: should contain exactly two elements.
- Integer: assumes only a user id, which is identical to the group id, and will be returned as `uid:gid`.
- Strings: If they include a colon, are returned as is; otherwise formatted as `name:name`, where *name* is assumed to be the user and group id.

3.5.2 Files and directories

There are shortcuts available for a few common tasks, which are more infrequently used in Dockerfiles, but otherwise applied by Docker-Map. Most of them in syntax and functionality correspond with the identical unix shell commands.

The command `mkdir()` returns a string for creating directories. By default, parent directories are created as necessary, which can be deactivated by setting `create_parent=False`. Additionally, a bash *if*-clause can be inserted to check first whether the directory already exists. This is not the default, but set with `check_if_exists=True`.

Commands generated by utility functions `chmod()` modify file system permission flags, `chown()` changes the owner, just like their corresponding unix commands. The *chmod* permissions can be written in any notation as accepted by the unix command line. The user name for *chown* is expanded to a `user:group` notation using `get_user_group()`. For removing files, `rm()` can be used for generating a command line.

By default `chmod()`, `chown()`, and `rm()` include the `-R` argument, i.e. they apply changes recursively. This behavior is changed by passing the optional keyword argument `recursive=False`.

A shortcut for combining `chmod()`, `chown()`, and `mkdir()` is `mkdir_chown()`: It generates a concatenated command for creating a directory `path` and applying file system ownership from `user_group` and permission flags from `permissions`. Both are not mandatory and skipped if set to `None`. The default for `permissions` is `ug=rwX,o=rX`. Note that in this function, `chmod()` and `chown()` are not recursive by default, but optional with setting `recursive=True`. Optionally, an *if*-clause can check whether the directory exists with the keyword argument `check_if_exists=True`; if it does, the other two functions *chmod* and *chown* are nevertheless applied.

For example an empty directory, available only to the user with id *2001*, is prepared with the following command:

```
df = DockerFile(...)
... #  Additional build commands
df.run(mkdir_chown('/var/lib/app', 2001, 'u=rwX,go='))
```

3.5.3 Miscellaneous

There are two utility functions for downloading files: `curl()` and `wget()`. Both have the URL as first argument, and an optional output file as second. Note that both programs need to be available in the base image, and that they behave differently when not provided with an output file parameter: *curl* prints the downloaded file to *stdout*, whereas *wget* attempts to detect the file name and stores it in the current directory.

Tip: A *Dockerfile* build can also download files with the `ADD` command.

Handling gzip-compressed tar archives (e.g. from downloads in Docker builds) can furthermore be supported with `targz()` and `untargz()`. Both have the archive name as the first argument. For `targz()`, specifying source files as second argument is obligatory, whereas `untargz()` has an optional destination argument, but will by default extract to the current directory.

3.6 Enhanced client functionality

The library comes with an enhanced client for some added functionality. Docker-Map is relying on that for managing container creation and startup. One part of the client is `DockerClientWrapper`, a wrapper around *docker-py*'s client; another is the application of container maps in form of `ContainerMap` instances to this client, which is handled by `MappingDockerClient`.

Since version 0.2.0 it is possible to use `MappingDockerClient` without `DockerClientWrapper`. The following paragraphs describe the added wrapper functionality. If you are not interested in that, you can skip to *Applying container maps*.

3.6.1 Wrapped functionality

In a few methods, the original arguments and behavior of *docker-py* has been modified in `DockerClientWrapper`:

Building images

On the build method `build()`, it is mandatory to give the new image a name (short example in *Building the Docker image*). If needed, add more tags by specifying `add_tags`. Optionally `add_latest_tag` can be set to `True` for tagging the image additionally with *latest*.

Whereas *docker-py* returns a stream, the wrapped method sends that stream to a log (see *Logging*) and returns the new image id, if the build has been successful. Unsuccessful builds return `None`.

Registry access

A login to a registry server with `login()` only returns `True`, if it has been successful, or `False` otherwise. Registry `pull()` and `push()` actions process the stream output using `push_log()`; they return `True` or `False` depending on whether the operation succeeded.

3.6.2 Added functionality

The following methods are not part of the original *docker-py* implementation:

Logging

Feedback from the service is processed with `push_log()`. The default implementation uses the standard logging system. Progress streams are sent using `push_progress()`, which by default is not implemented. Logs for a running container can be shown with `push_container_logs()`. Each message is prefixed with the container name.

Building from DockerFile and DockerContext

In order to build files directly from `DockerFile` and `DockerContext` instances, `build_from_file()` and `build_from_context()` are available. For details, see *Building images with DockerFile*.

Managing images and containers

On development machines, containers often have to be stopped, removed, and restarted. Furthermore, when repeatedly building images, there may be a lot of unused images around.

Calling `cleanup_containers()` removes all stopped containers from the remote host. Containers that have never been started are not deleted. `remove_all_containers()` stops and removes all containers on the remote. Use this with care outside of the development environment.

For removing images without names and tags (i.e. that show up as *none*), use `cleanup_images()`. Optionally, setting `remove_old` to `True` additionally removes images that do have names and tags, but not one with *latest*:

```
client.cleanup_images(remove_old=True)
```

All current container names are available through `get_container_names()`, for checking if they exist. Similarly `get_image_tags()` returns all named images, but in form of a dictionary with a name-id assignment.

Storing images and resources

The original implementations of `copy` (copying a resource from a container) and `get_image` (retrieving an image as a tarball) are available directly, but they return a stream. Implementations of `copy_resource()` and `save_image()` allow for writing the data directly to a local file. However, this has turned out to be very slow and may not be practical.

3.6.3 Applying container maps

This section provides some background information of the client functionality. The configuration and an example is further described in *Managing containers*.

Instances of MappingDockerClient are usually created with a map and a client. The former is an instance of ContainerMap, the latter is a Client object. Both initializing arguments are however optional and may be changed any time later using the properties maps:

```
from dockermap.api import DockerClientWrapper, MappingDockerClient

map_client = MappingDockerClient(container_map, DockerClientWrapper('unix://var/run/
↪docker.sock'))
```

Since version 0.2.0, also multiple maps and clients are supported. If exactly one map is provided, it is considered the default map. That one is always used when not specified otherwise in a command (e.g. create). Similarly, there can be a default client, which is used whenever a container map or container configuration does not explicitly state a different set of clients.

Clients are configured with ClientConfiguration objects, which are passed to the MappingDockerClient constructor:

```
from dockermap.api import ClientConfiguration, MappingDockerClient

clients = {
    'client1': ClientConfiguration('host1'),
    'client2': ClientConfiguration('host2'),
    ...
}
map_client = MappingDockerClient([container_map1, container_map2, ...],      #␣
↪Container maps as list, tuple or dict
                                 clients['client1'],                         # Default␣
↪client, optional
                                 clients=clients)                            # Further␣
↪clients
```

These clients are then used according to the *Clients* configuration on a container map. The default client can be referenced with the name __default__.

3.7 Managing containers

The concept of application containers has drastically increased in popularity over the past few years. Containerized services however need to be set up in a different way: All resources (e.g. ports and file systems) need to be shared explicitly; otherwise they are unavailable to the container or the host using them. Services that the application depends on need to be available when the container starts. This leads to some complexity in startup procedures that is often attempted to manage with scripts.

3.7.1 Container landscapes with ContainerMap

The implementation of ContainerMap aims to address aforementioned issues.

- It configures the creation and start of containers, including their dependencies. It pulls images, creates volumes, and configures networks as necessary.
- Shared resources (e.g. file systems, unix domain sockets) can be moved to shared volumes; permissions can be adjusted upon startup.

Container maps can be created empty and defined by code, updated from dictionaries, loaded from YAML, or combinations of those methods. Every map has a name, that is set on instantiation:

```python
from dockermap.api import ContainerMap

container_map = ContainerMap('new_map')
```

Structure

A `ContainerMap` carries the following main elements:

- `containers`: A set of container configurations.
- `volumes`: Shared volume aliases to be used by the container configurations, that can also be configured with common Docker options.
- `host`: Host volume paths, if they are mapped from the host's file system.
- `networks`: If the Docker version used supports it, allows for configuration of additional networks.
- `groups`: Container configurations can be grouped together, so that actions can be performed in common.

Their contents can be accessed like regular attributes, e.g.:

```python
container_map.containers.app1.binds = 'volume1'
container_map.volumes.volume1 = '/var/log/service'
container_map.host.volume1 = '/var/log/app1'
```

or in a dictionary-like syntax:

```python
container_map.containers['app1'].binds = 'volume1'
container_map.volumes['volume1'] = '/var/log/service'
container_map.host['volume1'] = '/var/log/app1'
```

Note: Elements of `containers` do not have to be instantiated explicitly, but are created upon their first access. For accessing only defined container configurations, see `get_existing`.

Additionally there are the following attributes:

- `name`: All created containers will be prefixed with this.
- `repository`: A prefix, that will be added to all image names, unless they already have one or start with `/` (i.e. are only local images).
- `default_domain`: The domain name that is set on new containers; it can be overridden by a client configuration. If none of the two are available, Docker's default is used.
- `set_hostname`: For specifying a new container's host name dependent on the container name (in the format `<client name>-<container name>`), this is by default set to `True`. If set to `False`, Docker automatically generates a new host name for each container.
- `use_attached_parent_name`: If you would like to re-use the same volume aliases for *Selectively sharing volumes* or apply *inheritance*, this changes the naming scheme of attached volume containers to include the name of their parent container.

Volumes

Typically Docker applications rely on finding shared files (e.g. working data, log paths) in a specific directory. The `volumes` of a container map assigns aliases to those elements. At the same time, volumes can be configured here including the following properties:

- `default_path`: The path that is used for this volume alias by default (i.e. unless overridden in a container configuration). This is the only property that is assigned when only a string is passed when creating this configuration.
- `driver`: Volume drive to use, in case this is a named Docker volume that is shared between containers. For host volumes this has no effect. The default is `local`.
- `driver_options`: Further options to pass to the driver on volume creation.
- `create_options`: Additional arguments for volume creation, that are not further connected to Docker-Map's functionality.
- `user`: On volume creation, ownership of the virtual path is set to this user/group with a `chown` command. For host volumes, this has no effect. If not set, the user of the container configuration is used that has this one set in `attaches`, if any.
- `permissions`: Similar to the `user`, permissions on the virtual path of the volume can be made through this setting, and is applied using `chmod` on container creation.

Depending on its purpose, this can be used for configuring volumes or just for assigning default volume paths to an alias:

```
container_map.volume1 = '/var/lib/my-app/data'
```

is equivalent to:

```
container_map.volume1 = VolumeConfiguration(default_path='/var/lib/my-app/data')
```

Host

The `host` is a single instance of `HostVolumeConfiguration`. This is very similar to `volumes`, but it defines paths on the host-side. Every alias used here should also be defined container-side in `volumes`.

Beside that, a `HostVolumeConfiguration` has the optional property `root`. If the paths are relative paths (i.e. they do not start with `/`), they will be prefixed with the *root* at run-time.

Usually paths are defined as normal strings. If you intend to launch multiple `instances` of the same container with different host-path assignments, you can however also differentiate them as a dictionary:

```
container_map.containers.app1.instances = 'instance1', 'instance2'
...
container_map.host.volume1 = {
    'instance1': 'config/instance1',
    'instance2': 'config/instance2',
}
```

Networks

Networks can be configured with the properties as specified in the Docker API docs. The `driver` is usually set to `bridge` for custom networks (and is therefore the default). Driver options can be added in `driver_options`. For any parameters not supported by this configuration, `create_options` can be used:

```
from dockermap.api import NetworkConfiguration

container_map.networks.network1 = NetworkConfiguration(internal=True,
                                                       driver_options={
```

```
                                                                        'com.docker.network.bridge.
↪enable_icc': 'false'
                                                                    })
```

Clients

Since version 0.2.0, a map can describe a container structure on a specific set of clients. For example, it is possible to run three application servers on a set of hosts, which are reverse-proxied by a single web server. This scenario would be described using the following configuration:

```
from dockermap.api import ClientConfiguration

clients = {
    'apps1': ClientConfiguration(base_url='apps1_host', interfaces={'private': '10.x.
↪x.11'}),
    'apps2': ClientConfiguration(base_url='apps2_host', interfaces={'private': '10.x.
↪x.12'}),
    'apps3': ClientConfiguration(base_url='apps3_host', interfaces={'private': '10.x.
↪x.13'}),
    'web1': ClientConfiguration(base_url='web1_host', interfaces={'private': '10.x.x.
↪21', 'public': '178.x.x.x'}),
}
apps_container_map.clients = 'apps1', 'apps2', 'apps3'
web_container_map.clients = 'web1'
```

The *interfaces* definition can later be used when specifying the address that a port is to be exposed on.

Clients specified within a container configuration have a higher priority than map-level definitions.

Container configuration

Container configurations are defined within `ContainerConfiguration` objects. They have the following properties:

Image

The `image` simply sets the image to instantiate the container(s) from. As usual, new containers are used from the image with the `latest` tag, unless explicitly specified using a colon after ithe image name, e.g. `ubuntu:16.10`. Using the `default_tag` property on the parent map, this becomes the new default tag. For example, if you usually tag all *development* images as `devel` and set `default_tag` accordingly, setting `image` to `image1` results in using the image `image1:devel`.

If `repository` is set on the parent `ContainerMap`, it will be used as a prefix to image names.

For example, if you have a local registry under *registry.example.com*, you likely do not want to name each of your images separately as `registry.example.com/image1`, `registry.example.com/image2`, and so on. Instead, just set the `repository` to `registry.example.com` and use image names `image1`, `image2` etc.

As an exception, any image with `/` in its name will not be prefixed. In order to configure the *ubuntu* image, set `image` to `/ubuntu` or `/ubuntu:16.10`.

If the image is not set at all, by default an image with the same name as the container will be attempted to use. Where applicable, it is prefixed with the `repository` or enhanced with `default_tag`.

Examples, assuming the configuration name is `app-server`:

image	repository	default_tag	Expanded image name
–	–	–	app-server:latest
image1			image1:latest
–	registry.example.com		registry.example.com/app-server:latest
image1			registry.example.com/image1:latest
–	–	devel	app-server:devel
image1			image1:devel
–	registry.example.com		registry.example.com/app-server:devel
image1			registry.example.com/image1:devel
/image1			image1:devel
image1:one			registry.example.com/image1:one
/image1:two			image1:two

Instances

If you plan to launch containers from the same image with an identical configuration, except for paths on the host system that are mapped to shared folders, these containers can be named as `instances`. The instance name is appended to the default container name on instantiation. If this property is not set, there is only one default instance.

Stop timeout

When stopping or restarting a container, Docker sends a `SIGTERM` signal to its main process. After a timeout period, if the process is still not shut down, it receives a `SIGKILL`. Some containers, e.g. database servers, may take longer than Docker's default timeout of 10 seconds. For this purpose `stop_timeout` can be set to a higher value.

Tip: This setting is also available on client level. The container configuration takes precedence over the client setting.

Where the Docker host supports it (API version >=1.25), and provided you are using version 2.x of the Docker Python library, this setting is also passed during container creation so that manually stopping a container (e.g. through CLI `docker stop <container>`) is making use of this setting as well. Otherwise this only applies when stopping or restarting containers through Docker-Map.

Stop signal

Not all applications handle `SIGTERM` in a way that is expected by Docker, so setting `stop_timeout` may not be sufficient. For example, PostgreSQL on a `SIGTERM` signal enters Smart Shutdown mode, preventing it from accepting new connections, but not interrupting existing ones either, which can lead to a longer shutdown time than expected.

In this case you can use a more appropriate signal, e.g. `SIGINT`. Set either the text representation (`SIGINT`, `SIGQUIT`, `SIGHUP` etc.) or the numerical constant (see the *signal* man page) in the property `stop_signal`. It will be considered during stop and restart actions of the container. As usual, `SIGKILL` will be used after, if necessary.

Where the Docker host supports it (API version >=1.21), this setting is also passed during container creation so that manually stopping a container (e.g. through CLI `docker stop <container>`) is making use of this setting. Otherwise this only applies when stopping or restarting containers through Docker-Map.

Shared volumes

Volume paths can be set in `shares`, just like the `VOLUME` command in the Dockerfile or the `-v` argument to the command line client. You do not need to specify host-mapped volumes here – this is what `binds` is for.

Volumes shared with the host

Volumes from the host, that are accessed by a single container, can be configured in one step:

```
container_map.containers.app1.binds = {'container_path': ('host_path', 'ro')}
```

For making the host volume accessible to multiple containers, it may be more practical to use an volume alias:

1. Create an alias in `volumes`, specifying the path inside the container.
2. Add the host volume path using the same alias under `host`.
3. Then this alias can be used in the `binds` property of one or more containers on the map.

Example:

```
container_map.volumes.volume1 = '/var/log/service'
container_map.volumes.volume2 = '/var/run/service'
container_map.host.volume1 = '/var/app1/log'
container_map.host.volume2 = '/var/app1/run'
# Add volume1 as read-write, make volume2 read-only.
container_map.containers.app1.binds = ['volume1', ('volume2', True)]
```

The definition in `host` is usually a list or tuple of `SharedVolume` instances.

For easier input, this can also be set as simple two-element Python tuples, dictionaries with each a single key; strings are also valid input, which will default to read-only-access (except `rw`).

The following are considered the same for a direct volume assignment (without alias), for read-only access:

```
container_map.containers.app1.binds = {'container_path': ('host_path', 'ro')}
container_map.containers.app1.binds = {'container_path': ('host_path', 'true')}
container_map.containers.app1.binds = [('container_path', 'host_path', True)]
container_map.containers.app1.binds = (['container_path', ('host_path', True)], )
```

Using aliases and two different forms of access, the following has an identical result:

```
container_map.containers.app1.binds = {'volume1': 'rw', 'volume2': True}
container_map.containers.app1.binds = ['volume1', ('volume2', True)]
container_map.containers.app1.binds = [['volume1'], ('volume2', 'ro')]
```

Note: Volume paths on the host are prefixed with `root`, if the latter is set and the container path does not start with a slash. This also applies to directly-assigned volume paths without alias.

Volumes shared with other containers

Inserting container names in `uses` is the equivalent to the `--volumes-from` argument on the command line.

You can refer to other containers names on the map, or names listed in the `attaches` property of other containers. When referencing other container names, this container will have access to all of their shared volumes; when referencing attached volumes, only the attached volume will be accessible. Either way, this declares a dependency of one container on the other.

Like `host`, input to `uses` can be provided as tuples, dictionaries, or single strings, which are converted into lists of `SharedVolume` tuples.

Selectively sharing volumes

There are multiple possibilities how a file system can be shared between containers:

- Assigning all containers the same host volume. This is the most practical approach for persistent working data.
- Sharing all volumes of one container with another. It is the most pragmatic approach for temporary files, e.g. pid or Unix sockets. However, this also implies access to all other shared volumes such as host paths.
- Specific volumes can be shared one by one. For example, a web application server communicating with its cache over Unix domain sockets needs access to the latter, but not the cache's data or configuration. In more recent releases (since API 1.21), Docker supports named volumes for this purpose. On older releases, Docker-Map emulates this behavior by creating an extra container that shares a single volume.

Volumes for selective sharing with other containers can be created using the `attaches` property. It refers to an alias in `volumes` in order to define a path and optionally the configuration. At the same time, this becomes the name of the volume (or extra container), and other container configurations can refer to it in the `uses` property.

Actually using `attaches` and `uses` is just like using the `-v` or `--volumes-from` on the command line, but this concept implies that one container *owns* the volume and provides data, a socket file, or something else there for other containers to use.

In the aforementioned emulation pattern, *attached* containers are by default automatically created and launched from a minimal startable base image *tianon/true*. They are also shared with the owning container:

```
container_map.volumes.volume1 = '/var/data1'
container_map.volumes.volume2 = '/var/more_data'
container_map.host.volume1 = '/var/app1/data1'
container_map.containers.app1.binds = 'volume1'
container_map.containers.app1.attaches = 'volume2'
...
# app2 inherits all shared volumes from app1
container_map.containers.app2.uses = 'app1'
# app3 only gains access to 'volume2'
container_map.containers.app3.uses = 'volume2'
```

Sharing data with other containers with non-superuser privileges usually requires permission adjustments. Setting `user` starts one more temporary container (based on *busybox*) running a `chown` command. Furthermore this sets the user that the current container is started with. Similarly for `permissions`, the temporary *busybox* container performs a `chmod` command on the shared container. If the client supports running local commands via a method `run_cmd`, instead of running the temporary container, `chmod` and `chown` will be run on the mounted volume path of the Docker host.

Linked containers

Containers on the map can be linked together (similar to the `--link` argument on the command line) by assigning one or multiple elements to `links`. As a result, the container gains access to the network of the referenced container. This also defines a dependency of this container on the other.

Elements are set as `ContainerLink` named tuples, with elements `(container, alias)`. However, it is also possible to insert plain two-element Python tuples, single-key dictionaries, and strings. If the alias is not set (e.g. because only a string is provided), the alias is identical to the container name, but without the name prefix of the *ContainerMap*.

Exposed ports

Containers may expose networking ports to other services, either to *Linked containers* or to a host networking interface. The `exposes` property helps setting the ports and bindings appropriately during container creation and start.

The configuration is set either through a list or tuple of the following:

- a single string or integer - exposes a port only to a linked container;
- a pair of string / integer values - publishes the exposed port (1) to the host's port (2) on all interfaces;
- a pair of string / integer values, followed by a string - publishes the exposed port (1) to the host's port (2) on the interface alias name (3), which is substituted with the interface address for that interface defined by the client configuration;
- additionally a fourth element - a boolean value - indicating whether it is an IPv6 address to be published. The default (`False`) is to use the IPv4 address from the client configuration of the interface alias in (3).

The publishing port, interface, and IPv6 flag can also be placed together in a nested tuple, and the entire configuration accepts a dictionary as input. All combinations are converted to `PortBinding` tuples with the elements `(exposed_port, host_port, interface, ipv6)`.

Examples:

```
## Exposes

clients = {
    'client1': ClientConfiguration({
        'base_url': 'unix://var/run/docker.sock',
        'interfaces': {
            'private': '10.x.x.x',  # Example private network interface IPv4 address
            'public: '178.x.x.x',   # Example public network interface IPv4 address
        },
        'interfaces_ipv6': {
            'private': '2001:a01:a02:12f0::1',  # Example private network interface␣
↪IPv6 address
        },
    }),
    ...
})

config = container_map.containers.app1
config.clients = ['client1']
config.exposes = [
    (80, 80, 'public'),            # Exposes port 80 and binds it to port 80 on the␣
↪public address only.
    (9443, 443),                   # Exposes port 9443 and binds to port 443 on all␣
↪addresses.
    (8000, 8000, 'private'),       # Binds port 8000 to the private network interface␣
↪address.
    8111,                          # Port 8111 will be exposed only to containers that␣
↪link this one.
    (8000, 80, 'private', True),   # Publishes port 8000 from the container to port 80␣
↪on the host under its private
```

```
                                              # IPv6 address.
]
```

Networking

Docker offers further options for controlling how containers communicate with each other. By default, it creates a new network stack of each, but it is also possible to re-use the stack of an existing container or disable networking entirely. The following syntax is supported by `network_mode`:

- `bridge` or `host` have the same effect as when used inside `host_config`. The former is the default, and creates a network interface connected to `docker0`, whereas the latter uses the Docker host's network stack.
- Similarly, `container:` followed by a container name or id reuses the network of an existing container. In this syntax, the container is assumed not to be managed by Docker-Map and therefore dependencies are not checked. The same applies for `/` followed by a container name or id.
- Setting it to the name of another container configuration (without the map name) will re-use that container's network. This declares a dependency, i.e. the container referred to will be created and started before the container that is re-using its network. Note that if there are multiple instances, you need to specify which instance the container is supposed to connect to in the pattern `<container name>.<instance name>`.
- `disabled` turns off networking for the container.

Starting with Docker API 1.21, there is also an additional `networks` property . Configured networks can be created and containers can be connected and disconnected during creation as well as at runtime.

For example, two containers can be connected in a network using the following setup:

```
from dockermap.api import ContainerConfiguration, NetworkConfiguration
container_map.networks.network1 = NetworkConfiguration(driver='bridge')
container_map.container1.networks = 'network1'
container_map.container2.networks = 'network2'
```

Starting either of the containers will automatically create the network before. Endpoints can also be configured in more detail. The argument order of parameters is * Network name, * a list of alias names on the network: `aliases`, * a list of linked containers: `links`, * the IPv4 address to use: `ipv4_address`, * the IPv6 address `ipv6_address`, * and a list of link-local IPs `link_local_ips`.

However, as all of the above are optional, they can also be declared explicitly:

> container_map.container1.networks = {'network1': {'ipv4': '172.17.0.5'}}

Lists as mentioned above are also accepted as single values on input and converted to a list automatically.

Note: The default network `bridge` is only implied if nothing is set for the container. This means that a container that has configured networks will **not** connect to `bridge` by default. This means that you need to add it if you would like a container to use it, e.g.:

```
container_map.container1.networks = {
    'network1': {'ipv4': '172.17.0.5'},
    'bridge': None,   # But it does not need explicit configuration.
}
```

Commands

By default every container is started with its pre-configured entrypoint and command. These can be overwritten in each configuration by setting `entrypoint` or `command` in `create_options`.

In addition to that, `exec_commands` allows for setting commands to run directly after the container has started, e.g. for processing additional scripts. The following input formats are considered:

- A simple command line is launched with the configured `user` of the container, or `root` if none has been set:

```
config.exec_commands = "/bin/bash -c 'script.sh'"
config.exec_commands = ["/bin/bash -c 'script.sh'"]
```

- A tuple of two elements is read as `command line, user`. This allows for overriding the user that launches the command. In this case, the command line can also be a list (executeable + arguments), as allowed by the Docker API:

```
config.exec_commands = [
    ("/bin/bash -c 'script1.sh'", 'root'),
    (['/bin/bash', '-c', 'script2.sh'], 'user'),
]
```

- A third element in a tuple defines when the command should be run. `dockermap.map.input.ExecPolicy.RESTART` is the default, and starts the command each time the container is started. Setting it to `dockermap.map.input.ExecPolicy.INITIAL` indicates that the command should only be run once at container creation, but not at a later time, e.g. when the container is restarted or updated:

```
from dockermap.map.input import ExecPolicy
config.exec_commands = [
    ("/bin/bash -c 'script1.sh'", 'root'),                         # Run␣
↪each time the container is started.
    (['/bin/bash', '-c', 'script2.sh'], 'user', ExecPolicy.INITIAL),   # Run only␣
↪when the container is created.
]
```

Inheritance

Container configurations can inherit settings from others, by setting their names in `extends`.

Example:

```
generic_config = container_map.containers.generic
generic_config.uses = 'volume1'
generic_config.abstract = True               # Optional - config is not used directly.
ext_config1 = container_map.containers.app1
ext_config1.extends = 'generic'
ext_config1.uses = 'volume2'                 # Actually uses ``volume1`` and␣
↪``volume2``.
ext_config2 = container_map.containers.app2
ext_config2.extends = 'generic'
ext_config2.uses = 'volume3'                 # Actually uses ``volume1`` and␣
↪``volume3``.
```

The behavior of value inheritance from other configurations is as follows:

- Values are overridden or merged in the order that they occur in `extends`. Extensions are followed recursively in this process.

- Simple values, e.g. image, are inherited from the other configurations and overridden in the extension.
- Single-value lists, e.g. those of clients or uses, are merged so that they contain the union of all values.
- Multi-value lists and dictionaries are merged together by their first value or their key, where applicable. For example, using the same local path in binds will use the last host path and read-only flag set in the order of inheritance. Similarly, create_options are merged so that they contain the union of all values, overriding identical keys in the extended configurations.

Note: Usually attached containers need to have unique names across multiple configurations on the same map. By default their naming on these containers follows the scheme <map name>.<attached volume alias>, which could become ambiguous when extending a configuration with attached volumes. When setting use_attached_parent_name to True, the naming scheme becomes <map name>.<parent container name>.<attached volume alias>, leading to unique container names again. In uses, you then need to refer to containers by <parent container name>.<attached volume alias>.

Example:

```
container_map.use_attached_parent_name = True
generic_config = container_map.containers.generic
generic_config.attaches = 'volume1'
ext_config = container_map.containers.app1
ext_config.extends = 'generic'
ext_config.uses = 'volume2'
ref_config = container_map.containers.test
ref_config.uses = ['app1.volume1', 'volume2']  # Now needs to specify the container␣
↪for attached volume.
```

Additional options

The properties create_options and host_config are dictionaries of keyword arguments. They are passed to the Docker Remote API functions in addition to the ones indirectly set by the aforementioned properties.

- The user that a container is launched with, inherited from the user configuration, can be overridden by setting user in create_options.
- Entries from volumes in create_options are added to elements of shares and resolved aliases from binds.
- Mappings on volumes_from in host_config override entries with identical keys (paths) generated from uses; non-corresponding keys are merged.
- Similarly, links keys set in host_config can override container links derived from links with the same name. Non-conflicting names merge.
- Containers marked with persistent set to True are treated like attached volumes: They are only started once and not removed during cleanup processes.

Start and create options can also be set via keyword arguments of create() and start(), in summary the order of precedence is the following:

1. Keyword arguments to the create() and start();
2. create_options and host_config;
3. and finally the aforementioned attributes from the ContainerConfiguration;

whereas single-value properties (e.g. user) are overwritten and dictionaries merge (i.e. override matching keys).

Note: Setting `start_options` has the same effect as `host_config`. The API version reported by the Docker client decides whether the recommended HostConfig dictionary is used during container creation (>= v1.15), or if additional keyword arguments are passed during container start.

Besides overriding the generated arguments, these options can also be used for addressing features not directly related to *Docker-Map*, e.g.:

```
config = container_map.containers.app1
config.create_options = {
    'mem_limit': '3g',   # Sets a memory limit.
}
config.host_config = {
    'restart_policy': {'MaximumRetryCount': 0, 'Name': 'always'},   # Unlimited␣
↪restart attempts.
}
```

Instead of setting both dictionaries statically, they can also refer to a callable. This has to resolve to a dictionary at run-time.

Note: It is discouraged to overwrite paths of volumes that are otherwise defined via `uses` and `binds`, as well as exposed ports as set via `exposes`. The default policy for updating containers will not be able to detect reliably whether a running container is consistent with its configuration object.

Input formats

On the attributes `extends`, `instances`, `shares`, `binds`, `attaches`, `uses`, `links`, `exposes`, `networks`, `exec_commands`, and `clients`, any input value that is not a list will be converted into one:

```
container_map.containers.app1.uses = 'volume1'
```

does the same as:

```
container_map.containers.app1.uses = ['volume1']
```

and:

```
container_map.containers.app1.uses = ('volume1',)
```

Additional conversions are made for `binds`, `attaches`, `uses`, `links`, `exposes`, `networks`, `exec_commands`, and `healthcheck`; each element (where applicable, i.e. each list item) is converted into a named tuple `SharedVolume`, `HostVolume`, `UsedVolume`, `ContainerLink`, `PortBinding`, `NetworkEndpoint`, `ExecCommand`, or `HealthCheck`.

This conversion takes places place, together with some input validation, if `clean()` is called directly or indirectly through any of `as_dict()`, `merge_from_dict()`, `merge_from_obj()`, or `update_from_obj()`. Since the merge-methods are also used before resolving container dependencies, this is done implicitly before invoking any command on container maps.

3.7.2 Creating and using container maps

A map can be initialized with or updated from a dictionary. Its keys and values should be structured in the same way as the properties of `ContainerMap`. There are two exceptions:

- Container names with their associated configuration can be, but do not have to be wrapped inside a `containers` key. Any key that is not `volumes`, `host`, `repository`, or `host_root` is considered a potential container name.
- The host root path `root` can be set either with a `host_root` key on the highest level of the dictionary, or by a `root` key inside the `host` dictionary.

For initializing a container map upon instantiation, pass the dictionary as the second argument, after the map name. This also performs a brief integrity check, which can be deactivated by passing `check_integrity=False` and repeated any time later with `check_integrity()`. In case of failure, it raises a `MapIntegrityError`.

A `MappingDockerClient` instance finally applies the container map to a Docker client. This can be a an instance of the Docker Remote API client. For added logging and additional functionality, using an instance of `DockerClientWrapper` is recommended. Details of these implementations are described in *Enhanced client functionality*.

Example

This is a brief example, given a web server that communicates with two app instances of the same image over unix domain sockets:

```python
from dockermap.api import ContainerMap

container_map = ContainerMap('example_map', {
    'repository': 'registry.example.com',
    'host_root': '/var/lib/site',
    'web_server': { # Configure container creation and startup
        'image': 'nginx',
        # If volumes are not shared with any other container, assigning
        # an alias in "volumes" is possible, but not necessary:
        'binds': {'/etc/nginx': ('config/nginx', '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': 'config/app1',
            'instance2': 'config/app2',
        },
        'app_data': {
            'instance1': 'data/app1',
            'instance2': 'data/app2',
        },
    },
})
```

This example assumes you have two images, `registry.example.com/nginx` for the web server and `registry.example.com/app` for the application server (including the app). Inside the `nginx` image, the working user is assigned to the group id `2000`. The app server is running with a user that has the id `2000`.

Creating a container with:

```
from dockermap.api import DockerClientWrapper, MappingDockerClient

map_client = MappingDockerClient(container_map, DockerClientWrapper('unix://var/run/
↪docker.sock'))
map_client.create('web_server')
```

results in the following actions:

1. Dependencies are checked. `web_server` uses `app_server_socket`, which is attached to `app_server`. Consequently, `app_server` will be processed first.
2. `app_server_socket` is created. The name of the new container is `example_map.app_server_socket`.
3. Two instances of `app_server` are created with the names `example_map.app_server.instance1` and `example_map.app_server.instance2`. Each instance is assigned a separate path on the host for `app_data` and `app_config`. In both instances, `app_config` is a read-only volume.
4. `web_server` is created with the name `example_map.web_server`, mapping the host path `/var/lib/site/config/nginx` as read-only. Ports 80 and 443 are exposed.

Furthermore, on calling:

```
map_client.start('web_server')
```

1. Dependencies are resolved, just as before.
2. `example_map.app_server_socket` is started, so that it can share its volume.
3. Temporary containers are started and run `chown` and `chmod` on the `app_server_socket` volume. They are removed directly afterwards.
4. `example_map.app_server.instance1` and `example_map.app_server.instance2` are started and gain access to the volume of `example_map.app_server_socket`.
5. `example_map.web_server` is started, and shares the volume of `example_map.app_server_socket` with the app server instances. Furthermore it maps exposed ports 80 and 443 to all addresses of the host, making them available to public access.

Both commands can be combined by simply running:

```
map_client.startup('web_server')
```

3.8 Container actions

This section explains what actions are implemented for managing containers in addition to the brief example in *Applying container maps*, as soon as *Managing containers* are set up.

3.8.1 Basic container commands

`MappingDockerClient` provides as set of configurable commands, that transform the container configurations and their current state into actions on the client, along with keyword arguments accepted by *docker-py*. By default it supports the following methods:

- `create()` resolves all dependency containers to be created prior to the current one. First, *attached* volumes are created (see *Selectively sharing volumes*) of the dependency containers. Then the client creates dependency containers and the requested container. Existing containers are not re-created.
- Similarly, `start()` first launches dependency containers' *attached* volumes, then dependencies themselves, and finally the requested container. *Persistent* and *attached*, containers are not restarted if they have exited.
- `restart()` only restarts the selected container, not its dependencies.
- `stop()` stops the current container and containers that depend on it.
- `remove()` removes containers and their dependents, but does not remove attached volumes.
- `startup()`, along the dependency path,
  - removes containers with unrecoverable errors (currently codes `-127` and `-1`, but may be extended as needed);
  - creates missing containers; if an attached volume is missing, the parent container is restarted;
  - and starts non-running containers (like *start*).
- `shutdown()` simply combines `stop()` and `remove()`.
- `update()` and `run_script()` are discussed in more detail below.

3.8.2 Updating containers

`update()` checks along the dependency path for outdated containers or container connections. In more detail, containers are removed, re-created, and restarted if any of the following applies:

- The image id from existing container is compared to the current id of the image as specified in the container configuration. If it does not match, the container is re-created based on the new image.
- Linked containers, as declared on the map, are compared to the current container's runtime configuration. For legacy container links, if any container is missing or the linked alias mismatches, the dependent container is re-created and restarted. If used with configured container networks, only the network endpoint is re-created.
- The virtual filesystem path of attached containers and other shared volumes is compared to dependent containers' paths. In case of a mismatch, the latter is updated.
- The environment variables, command, and entrypoint of the container are compared to variables set in `create_options`. If any of them are missing or not matching, the container is considered outdated.

- Exposed ports of the container are checked against `exposes`. If any ports are missing or configured differently, this also causes a container update.
- The following container limits are checked against the host configuration of the currently-running container:
  - `BlkioWeight`
  - `CpuPeriod`
  - `CpuQuota`
  - `CpuShares`
  - `CpusetCpus`
  - `CpusetMems`
  - `Memory`
  - `MemoryReservation`
  - `MemorySwap`
  - `KernelMemory`

  If the Docker API supports it (version >= 1.22), changes will be applied to a running container (i.e. without the need to re-create the container). Due to a limitation in the Docker SDK for Python, existing limits can only be modified by not be removed (or set to zero). Where necessary, the container is reset unless `skip_limit_reset=True` is passed in as a keyword argument.

The keyword argument `force_update` takes a single name or list of container configuration names to force the update on. This can be useful in cases where the need for re-creating a container cannot be detected automatically.

Post-start commands in `exec_commands` are checked if they can be found on a running container, matching command line and user. If not, the configured command is executed, unless `dockermap.map.input.ExecPolicy.INITIAL` has been set for the command. By default the entire command line is matched. For considering partial matches (i.e. if the command in the process overview gets modified), you can set `check_exec_commands` to `dockermap.map.input.CmdCheck.PARTIAL`. Setting it to `dockermap.map.input.CmdCheck.NONE` deactivates this check entirely.

Networks in `networks` are tested if they exist and if the endpoint id matches. (Re-)connecting as necessary does not require a container restart.

For ensuring the integrity, all missing containers are created and started along the dependency path. In order to see what defines a dependency, see *Volumes shared with other containers* and *Linked containers*.

Additional keyword arguments to the `start` and `create` methods of the client are passed through; the order of precedence towards the `ContainerConfiguration` is further detailed in *Additional options*. Example:

```
map_client.start('web_server', restart_policy={'MaximumRetryCount': 0, 'Name': 'always
↪'})
```

For limiting effects to particular *Instances* of a container configuration, all these methods accept an `instances` argument, where one or multiple instance names can be specified. By implementing a custom subclass of `BasePolicy`, the aforementioned behavior can be further adjusted to individual needs.

Note that `MappingDockerClient` caches names of existing containers and images for speeding up operations. The cache is flushed automatically when the `policy_class` property is set. However, when changes (e.g. creating or removing containers) are made directly, the name cache should be reset with `refresh_names()`.

Besides aforementioned methods, you can define custom container actions such as `custom` and run the using `call()` with the action name as the first argument. For this purpose you have to implement a policy class with a method `custom_action` with the first arguments *container map name*, *container configuration name*, and *instances*. Further keyword arguments are passed through.

3.8.3 Running scripts

The default client also implements a `run_script()` action. Its purpose is to run a script or single command inside a container and automatically perform the necessary creation, start, and cleanup, along with dependencies. Usage is slightly different from the other actions: Container configuration name and map name are the first two arguments – as usual – but the third is only one optional instance name. Additionally, the method supports the following optional arguments:

- `script_path`: This may either be a file or a directory on the Docker host. If it points to a file, this will be assumed to be the script to run via the command. The parent directory will be available to the container, i.e. all other files in the same directory as the script. If `entrypoint` and `command_format` describe a self-contained action that does not require a script file, you can still point this to a path to include more files or write back results.
- `entrypoint`: Entrypoint of the script runtime, e.g. `/bin/bash`.
- `command_format`: Just like `command` for a container, but any occurrence of a `{script_path}` variable is replaced with the path inside the container. This means that if `script_path` points to a script file `/tmp/script.sh`, the command will be formatted with `/tmp/script_run/test.sh` (prefixed with the path specified in `container_script_dir`). If it points to a directory, simply `container_script_dir` will be used in place of script path.
- `wait_timeout`: Maximum time to wait before logging and returning the container output. By default the waiting time set up for the container `stop_timeout` or for the client `timeout` is used.
- `container_script_dir`: Path to run the script from inside the container. The default is `/tmp/script_run`.
- `timestamps` and `tail` are simply passed through to the `logs` command of the *docker-py* client. They can be used to control the output of the script command.
- `remove_existing_before`: Whether to remove containers with an identical name if they exist prior to running this command. By default, an existing container raises an exception. Setting this to `True` can be a simple way to recovering repeatable commands that have run into a timeout error.
- `remove_created_after`: Whether to remove the container instance after a successful run (i.e. not running into a timeout), provided that it has been created by this command. This is the default behavior, so set this to `False` if you intend to keep the stopped container around.

The `run_script()` method returns a dictionary with the client names as keys, where the script was run. Values are nested dictionaries with keys `log` (the *stdout* of each container) the `exit_code` that the container returned, and the temporary container id that had been created. In case the `wait` command timed out, the container logs and exit code are not available. In that case, the nested dictionary contains the `id` of the container (which still exists) and a message in `error`.

Containers that were created in the course of running the script are also stopped and removed again, unless waiting timed out or `remove_created_after` was set to `False`. If the container of the configuration exists prior to the setup attempt and `remove_existing_before` is not set to `True`, the script will not be run. In that case a `ScriptActionException` is thrown. In order to

Script examples

For running a bash script, set the executable bit on the file in your local path, and run it:

```
map_client.run_script('test_container',
                      script_path='/tmp/test_path/test_script.sh',
                      entrypoint='/bin/bash',
                      command_format=['-c', '{script_path}'])
```

Assuming you have a *Redis* image and container with access to the socket in /var/run/redis/cache.sock, you can flush the database using:

```
map_client.run_script('redis_client',
                      entrypoint='redis-cli',
                      command_format=['-s', '/var/run/redis/cache.sock', 'flushdb'])
```

Importing a *PostgreSQL* database to a server accessed via /var/run/postgresql/socket, from a file stored in /tmp/db_import/my_db.backup, can be performed with:

```
map_client.run_script('postgres_client',
                      script_path='/tmp/db_import',
                      entrypoint='pg_restore',
                      command_format=['-h', '/var/run/postgresql/socket',
                                      '-d', 'my_db', '{script_path}/my_db.backup']
```

Note: In case files cannot be found by the script or command, check if ownership and access mode match the container user.

3.8.4 Options

Aforementioned commands support a set of options, which are processed by different elements of the policy framework. They can be modified in custom implementations, or changed by passing in keyword arguments.

- `remove_existing_after` (Actions: `script`; Default: `True`): Usually when the script action creates containers, it cleans up after. If you want to keep the containers, you can set this to `False`.
- `remove_existing_before` (Actions: `script`; Default: `False`): Containers by the script action are created new, and if they exist it raises an error. Setting this to `True` removes any container that exists before creating new ones.
- `remove_persistent` (Actions: `remove`, `shutdown`; Default: `True`): When removing containers, by default all configurations are considered. If this is set to `False`, configurations marked as `persistent` are skipped.
- `remove_attached` (Actions: `remove`, `shutdown`; Default: `False`): Attached containers serve for volume sharing on Docker versions prior to native volume support. By default they are not removed during `remove` and `shutdown` actions, but can optionally be included setting this to `True` .
- `pull_all_images` (Actions: `pull`; Default: `True`): The `pull` action attempts to download all images of the input container configurations, also updating existing ones with identical tags. If this is set to `False`, only missing tags are pulled from the registry.
- `pull_before_update` (Actions: `update`; Default: `False`): Before an `update` operation, images of configured containers can optionally be pulled before detecting changes by setting this to `True`.
- `pull_insecure_registry` (Actions: `pull`, `update`; Default: `False`): Docker by default requires a registry to be published on a TLS encrypted connection (i.e. a HTTPS url), and the presented certificates to be trusted by the client (i.e. the Docker host). This is good practice and highly recommended; however, as Docker still offers the possibility to allow insecure connections, setting this flag to `True` makes use of it.
- `prepare_local` (Actions: `create`, `startup`, `update`, `script`; Default: `True`):
- `nonrecoverable_exit_codes` (Actions: `update`; Default: `(-127, -1)`): Exit codes to assume that a container cannot be simply restarted, so that it has to be removed and re-created.

- `force_update` (Actions: `update`; Default: `None`): A string or list of container configurations that should be updated without checking. This can be used in case a change in the container configuration cannot be detected reliably.
- `skip_limit_reset` (Actions: `update`; Default: `False`): Due to a limitation of the current Docker API, limits of containers can be updated but not removed. This may lead to the need to re-create the container. If this is not desired, set this to `True`.
- `update_persistent` (Actions: `update`; Default: `False`): Whether to remove containers where configurations are marked as `persistent`.
- `check_exec_commands` (Actions: `update`; Default: `CmdCheck.FULL`): How to check the command of a running container against the configuration. By default performs to match the full command, but can be set to `CmdCheck.PARTIAL` for a partial lookup.
- `restart_exec_commands` (Actions: `restart`; Default: `False`): When a container is restarted and this is set to `True`, all configured exec commands are also restarted.

3.9 Container configurations in YAML

YAML (YAML Ain't Markup Language) files are often easier to write than Python dictionaries, and provide a good possibility to separate code from configuration. Container maps can be maintained in and loaded from YAML files. The contents are represented as a Python dictionary, and therefore, the configuration structure is identical.

3.9.1 YAML elements

When used according to the full specification, YAML is a very feature-rich and powerful language. This is only a quick introduction to the syntactical elements of YAML, as far as relevant for container maps:

- YAML elements can be structured in a hierarchy, similar to other markup languages. Just like in Python, the hierarchy level is defined by outline indentation.
- Every line without any prefix is a key-value pair `key:  value`, and read as items of an associative array (a dictionary in Python). An indented key indicates a nested structure.
- Lines prefixed with a dash `-` followed by a space represent items of a list.
- Most data types are implicit. For example, you do not need to quote strings, unless they consist of only numbers and a dot and therefore could be read as integer or float. When in doubt (e.g. for version numbers), you should quote them or prefix with the tag `!!str`.
- Strings are trimmed (unless within quotes); in a dictionary for example, it does not matter how much space there is between the key and the value.
- Lists and dictionaries can also be written in inline-syle in JSON syntax: Curly brackets represent a associative array (dictionary), square brackets a list.

For a more comprehensive reference, the Wikipedia article provides a good overview. The YAML specification also has detailed examples. There is also a type list, which decribes most important data types.

Example

The *Example* map can be more easily written as:

```
repository: registry.example.com
host_root: /var/lib/site
web_server:
  image: nginx
  binds:
    /etc/nginx:
    - config/nginx
    - 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: config/app1
    instance2: config/app2
  app_data:
    instance1: data/app1
    instance2: data/app2
```

Note: It is possible to write nested lists in YAML, either in JSON notation, e.g.

```
...
exec_commands:
- [['/bin/bash', '-c', 'script.sh'], 'root']
```

or described in YAML syntax

```
...
exec_commands:
-
  -
    - /bin/bash
    - -c
    - script.sh
  - root
```

A configuration of clients, such as briefly described in *Clients*, would be written in the following format:

```
apps1:
  base_url: apps1_host
  interfaces:
    private: 10.x.x.11
apps2:
  base_url: apps2_host
  interfaces:
    private: 10.x.x.12
apps3:
  base_url: apps3_host
  interfaces:
    private: 10.x.x.13
web1:
  base_url: web1_host
  interfaces:
    private: 10.x.x.21
    public: 178.x.x.x
```

3.9.2 Importing YAML maps

The easiest way to generate a `ContainerMap` from a YAML file is `load_map_file()`:

```
from dockermap.map import yaml
map = yaml.load_map_file('/path/to/example_map.yaml')
```

By default the map will be named according to a `name` element on the root level of the map; this can be overwritten, e.g.:

```
map = yaml.load_map_file('/path/to/example_map.yaml', 'apps')
```

The initial integrity check can be skipped by passing `check_integrity=False`.

If your YAML structure is not a file, but a stream, you can use `load_map()`. It takes a buffer as first argument; additional arguments are identical to `load_map_file`.

There are in total three ways to assign a name to a map during the import, in the following order of priority:

1. The name passed as a keyword argument in `load_map_file()` or `load_map()`.
2. The base file name without extension from `load_map_file()`, if an empty string is passed as the `name` argument.
3. An extra `name` element on the root level of the map.

3.9.3 Importing clients

When using multiple clients, where client-specific variables (URLs, network addresses etc.) are needed, you may also choose to store client configurations in a YAML file. It can be imported using:

```
clients = yaml.load_clients_file('/path/to/example_clients.yaml')
```

If you implement your own client configuration (especially useful if you implement a custom client), you can pass the class as second argument. By default, a dictionary of client names with associated `ClientConfiguration` objects is returned.

3.9.4 User and environment variables

As YAML allows for definition of custom tags, `!path` has been added for indicating variables that are supposed to be expanded upon import. This is done using `os.path.expandvars` and `os.path.expanduser` (in that order). The aforementioned example's `host_root` entry also could also be defined as:

```
host_root: !path $SITE_ROOT
```

When the tag is applied to a list or associative array, nested elements are also expanded on their first level of sub-elements:

```
host: !path
  web_config: $CONFIG_PATH/nginx
  app_config: !path
    instance1: $CONFIG_PATH/app1
    instance2: $CONFIG_PATH/app2
```

3.9.5 Lazy resolution of variables

The default implementation of `!path` resolves variables as soon as they are instantiated. If this is not intended, you can use the `!path_lazy` tag instead. Then the variables will not be resolved to their current values until they are used for the first time. This option is available on the elements listed under *Lazy resolution of variables*.

This may have little practical relevance for paths provided in environment variables, since these are usually set before the application starts. It may however be useful if you extend the YAML parser with your own tags, that resolve variables at run-time.

3.10 Advanced library usage

Docker-Map can be used within Python applications directly, but also be used as a base implementation for other libraries. This section covers some areas that may be relevant when implementing enhancements, like Docker-Fabric.

3.10.1 Implementing policies

Before version 0.7.0, policies compared container maps to the current state on the Docker client, and performed changes directly. In later versions, implementations of `BasePolicy` only define a few guidelines, such as how containers are named, how image names are resolved, and which client objects to use:

- `get_dependencies()` and `get_dependents()` return the dependency path of containers for deciding in which order to create, start, stop, and remove containers;
- `cname()`, `aname()`, `nname()`, `image_name()`, `get_hostname()` generate inputs for aforementioned functions. They can be overridden separately.

3.10.2 Changing behavior

Operations are performed by a set of three components:

- So-called state generators, implementations of `AbstractStateGenerator`, determine the current status of a container. They also establish if and in which the dependency path is being followed. Currently there are four implementations:

  - `SingleStateGenerator` detects the basic state of a single container configuration, e.g. existence, running, exit code.
  - `DependencyStateGenerator` is an extension of the aforementioned and used for forward-directed actions such as creating and starting containers, running scripts etc. It follows the dependency path of a container configuration, i.e. detecting states of a dependency first.
  - `UpdateStateGenerator` is a more sophisticated implementation of `DependencyStateGenerator`. In addition to the basic state it also checks for inconsistencies between virtual filesystems shared between containers and differences to the configuration.
  - `DependentStateGenerator` also detects the basic state of containers, but follows the reverse dependency path and is therefore used for stopping and removing containers.
- Action generators, implementations of `AbstractActionGenerator`, transform these states into planned client actions. There is one action generator implementation, e.g. `CreateActionGenerator` aims to create all containers along the detected states that do not exist.
- The runners perform the planned actions the client. They are implementations of `AbstractRunner` and decide how to direct the client to applying the container configuration, i.e. which methods and arguments to use. Currently there is only one implementation: `DockerClientRunner`.

The instance of `MappingDockerClient` decides which elements to use. For each action a pair of a state generator and action generator is configured in `generators`. `runner_class` defines which runner implementation to use.

3.10.3 Lazy resolution of variables

Container maps can be modified at any time, but sometimes it may be more practical to defer the initialization of variables to a later point. For example, if you have a function `get_path(arg1, keyword_arg1='kw1', keyword_arg2='kw2')`, you would usually assign the result directly:

```
container_map.host.volume1 = get_path(arg1, keyword_arg1='kw1', keyword_arg2='kw2')
```

If the value is potentially not ready at the time the container map is being built, the function call can be delayed until `volume1` is actually used by a container configuration. In order to set a value for lazy resolution, wrap the function and its arguments inside `dockermap.functional.lazy` or `dockermap.functional.lazy_once`. The difference between the two is that the latter stores the result and re-uses it whenever it is accessed more than once, while the former calls the function and reproduces the current value on every use:

```
from dockermap.functional import lazy
container_map.host.volume1 = lazy(get_path, arg1, keyword_arg1='kw1', keyword_arg2=
↪'kw2')
```

or:

```
from dockermap.functional import lazy_once
container_map.host.volume1 = lazy_once(get_path, arg1, keyword_arg1='kw1', keyword_
↪arg2='kw2')
```

Serialization issues

In case of serialization, it may not be possible to customize the behavior using aforementioned lazy functions. Provided that the input values can be represented by serializable Python types, these types can be registered for pre-processing using `register_type()`.

For example, if a library uses MsgPack for serializing data, you can represent a value for serialization with:

```
from msgpack import ExtType

MY_EXT_TYPE_CODE = 1
...
container_map.host.volume1 = ExtType(MY_EXT_TYPE_CODE, b'info represented as bytes')
```

ExtType is supported by MsgPack's Python implementation, and therefore as long as the byte data carries all information necessary to reproduce the actual value, no additional steps are necessary for serialization. During deserialization, you could usually reconstruct your original value by writing a simple function and passing this in `ext_hook`:

```
def my_ext_hook(code, data):
    if code == MY_EXT_TYPE_CODE:
        # This function should reconstruct the necessary information from the␣
↪serialized data.
        return my_info(data)
    return ExtType(code, data)
```

This is the preferred method. If you however do not have access to the loading function (e.g. because it is embedded in another library you are using), you can slightly modify aforementioned function, and register ExtType for late value resolution:

```
from dockermap.functional import register_type

def my_ext_hook(ext_data):
    if ext_data.code == MY_EXT_TYPE_CODE:
        return my_info(ext_data.data)
    raise ValueError("Unexpected ext type code.", ext_data.code)

register_type(ExtType, my_ext_hook)
```

Note that you have to register the exact type, not a superclass of it, in order for the lookup to work.

Pre-resolving values

Aforementioned type registry is limited to values as listed in *Availability*. Additionally it may be difficult to detect errors in the configuration beforehand. In case the data can be pre-processed at a better time (e.g. after deserialization, in a configuration method), the method `dockermap.funcitonal.resolve_deep()` can resolve a structure of lists and dictionaries into their current values.

Rather than registering types permanently, they can also be passed to that function for temporary use, e.g.:

```
from dockermap.functional import expand_type_name, resolve_deep

# assume aforementioned example of my_ext_hook

resolve_dict = {expand_type_name(ExtType): my_ext_hook}
map_content = resolve_deep(deserialized_map_content, types=resolve_dict)
```

Availability

Lazy value resolution is available at the following points:

- On container maps:
  - the main `repository` prefix;

  - paths for all `volumes` aliases;
  - the host volume `root` path;
  - and all `host` volume paths.
- Within container configurations:
  - the `user` property;
  - host ports provided in the `exposes`, but not for the exposed port of the container (i.e. the first item of the tuple);
  - elements of `create_options` and `start_options`;
  - items of `binds`, if they are not volume aliases, i.e. they directly describe container volume and host path.
  - command line and user defined in each element of `exec_commands`;
  - elements listed in `shares`;
  - and on the network endpoint configurations in `networks`.
- On network configurations:
  - the values of `driver_options`,
  - and the values of `create_options`.
- On client configuration: For addresses in `interfaces`.

3.11 Change History

3.11.1 1.0.0

- Implemented update of container settings (i.e. memory limit, cpu shares) without a need for resetting the container, where supported by more recent Docker hosts.
- Version-dependent client features such as volumes and networks have been consolidated into a dictionary-type property `features`.
- Added support for the Docker SDK for Python 2.x. Version 1.x is still supported. Using 2.x and a sufficiently recent Docker host adds the following functionality:
  - The `restart_policy` setting in the host config can be updated at runtime.
  - `stop_timeout` is passed to the container on creation. Whereas Docker-Map has been supporting this setting for a while now, this will also apply when manually stopping a container (e.g. through the command line).
- Added `healthcheck`: Adds a health check command to containers, if the Docker host and the API client supports it.
- `start_delay`: Simply adds a delay of *x* seconds after a container has been started. Can be used on older Docker hosts to prevent race conditions during container linking.
- Added Dockerfile properties `labels`, `shell`, `stopsignal`, and `healthcheck`. There is however no verification if the Docker host supports these commands.
- Added serialization: Container maps now have a method `as_dict()`, that return all contents as a nested Python dictionary structure. Passing this in as the `initial` argument of `ContainerMap` yields a copy of the original map.

- Input validation and conversion is no longer performed immediately when configuration properties are being set, but deferred to a later step. This will likely not have any practical implications for most users, since conversion still happens automatically before merge operations and before the configuration is being used. It can also be invoked by calling `clean()`.
- All tuple input lists from `dockermap.map.input` now accept dictionaries with their named parameters as input.
- For consistency with other properties, the order of arguments in the `HostConfig` tuple have been changed. It is not `path`, `host_path`, `readonly` (optional).

3.11.2 0.8.1

- Fixed Python 3 compatibility.
- Minor internal refactoring.

3.11.3 0.8.0

- Restart uses the configured stop signal for the container.
- Added option to restart exec commands on container restart.
- The result (i.e. the id) of exec commands is now returned by the runner.

3.11.4 0.8.0rc1

- Added checks on configured ip addresses and link-local ips. Additional general improvements to container network endpoint check against configuration.
- Added checks on volume driver and driver options.
- After an unsuccessful stop attempt, added another wait period so that the container has time to process the `SIGKILL` signal issued by Docker.
- Process a single `MapConfigId` as valid input.
- Moved `CmdCheck` flags to `input` module.

3.11.5 0.8.0b5

- The `pull_images()` action also pulls present images by default (i.e. updates them from the registry). This was optional before, and can be prevented by passing `pull_all_images=False`, only pulling missing image tags.
- Internal cleanup of converting input into configuration ids.

3.11.6 0.8.0b4

- Included main process id in state data, so that implementations can detect a container restart more easily.
- Handling deprecation of the `force` argument when tagging images in newer Docker releases. The tag is added automatically depending on the detected API version.
- Fixed update check of container network mode referring to another container.

- Additional minor bugfix from previous prereleases.

3.11.7 0.8.0b3

- Added `volumes`: Where the Docker host supports it, volumes can now be configured with additional properties such as driver and options. The original workaround of Docker containers sharing anonymous volumes no longer applies in this case.
- The default path of volumes in `attaches` volumes can now be defined, by using a dictionary or list of tuples. They no longer have to (but still can) be set in `volumes`.
- Where the Docker host supports named volumes, container-side paths of `uses` items can be overridden, provided that they are referring to attached volumes created through another container.
- Removed `clients` property from `ContainerConfiguration`. It caused too much complexity in responding to supported client features. In addition, it was likely to break dependency paths. `clients` is however still available.

3.11.8 0.8.0b2

- `MappingDockerClient` now wraps all exceptions so that partial results, i.e. actions that already have been performed on clients. It raises a `ActionRunnerException`, which provides information about the client and action performed, partial results through `results()`, but also the possibility to re-trigger the original traceback using `reraise()`.
- Similarly, direct calls to the utility client `DockerClientWrapper`, such as `cleanup_containers()` now return a `PartialResultsError`.
- Added `signal()` method to client.
- Images have been integrated into the dependency resolution. Images of a container and all of its dependencies can now be pulled with the new command `pull_images()`.
- Authentication information for the Docker registry can now be added to `dockermap.map.config.client.ClientConfiguration.auth_configs` and are considered during login and image pull actions.
- Added a built-in group `__all__`, that applies to all containers or even all configured maps on `MappingDockerClient`.
- Several adaptions which makes it easier for programs and libraries using the API to evaluate changes.
- More fixes to image dependency check, so that `cleanup_images()` now works reliably. Removals can also be forced where applicable.
- Implemented CLI, missing from 0.8.0b1.
- Various bugfixes from 0.8.0b1.

3.11.9 0.8.0b1

- Added `groups`: Generally an action (e.g. startup of containers) can now be run at once on multiple items. In order to make input easier, groups can be added to a map that refers to multiple configurations. Dependencies that multiple items have in common will only be followed once.
- Added forced update: Not all differences between the container configuration and an existing instance can be detected automatically. A parameter `force_update` can now trigger an update of particular containers.

- Added `networks`: Docker networks can now be configured on a map. Referring to them in the property `networks` from one or multiple container configurations will create them automatically. The former `network` setting has been renamed to `network_mode` for disambiguation.

3.11.10 0.7.6

- More sensible solution of Issue #15, not changing user-defined link aliases. Doing so could cause name resolution issues.

3.11.11 0.7.5

- Minor fixes for compatibility with newer Docker hosts.
- Followup fixes from Issue #15.

3.11.12 0.7.4

- Fixed case where `exec_create` does not return anything, as when commands are started immediately (e.g. the CLI, Issue #17).
- Improved accuracy of comparing the container command from the configuration with the container inspection info.
- Added parser for CLI `top` command, as needed for inspecting exec commands.

3.11.13 0.7.3

- Fixed command line generator for case where `cmd` is used as a keyword argument (Issue #16).

3.11.14 0.7.2

- Fixed recursive dependency resolution order.
- Setting an alias name is always optional for container links, even if `ContainerLinks` tuple is used directly.

3.11.15 0.7.1

- Added `version` method to command line generator.
- Internal refactoring: Moved configuration elements to individual modules. If you get any import errors from this update, please check if you are using convenience imports such as `from dockermap.api import ContainerMap` instead of the modules where the classes are implemented.
- Fixed `ContainerMap.containers` attribute access to work as documented.

Note: The default iteration behavior has also changed. Similar to `ContainerMap.host` and `ContainerMap.volumes`, it generates items. Before iteration was returning keys, as usual for dictionaries.

- Fixes for use of alternative client implementations (e.g. CLI, Issue #12).

- Fixed `link` argument for command line generator (Issue #13).
- Added replacement for invalid characters in generated host names (Issue #15).

3.11.16 0.7.0

- Refactoring of policy framework. The monolithic client action functions have been divided into separate modules for improving maintainability and testing. This also makes it easier to add more functionality. A few minor issues with updating containers and executing commands were resolved during this change.
- Added an experimental command line generator.

3.11.17 0.6.6

- Added evaluation of `.dockerignore` files.
- Several bugfixes from *0.6.6b1*.

3.11.18 0.6.6b1

- Added arguments to set additional image tags after build.
- Added `default_tag` property to container maps.
- Minor refactoring. Possibly breaks compatibility in custom policy implementations:
  - `dockermap.map.policy.cache.CachedImages.reset_latest` has been renamed to `reset_updated()`.
  - `ensure_image()` argument `pull_latest` has been renamed to `pull`.
  - `dockermap.map.policy.update.ContainerUpdateMixin.pull_latest` has been renamed to `pull_before_update`.
  - `dockermap.map.policy.base.BasePolicy.iname` has been renamed to `image_name()` and changed order of arguments for allowing defaults.

3.11.19 0.6.5

- Better support for IPv6 addresses. Added `ipv6` flag to port bindings and `interfaces_ipv6` property to client configuration.
- Command elements are converted into strings so that Dockerfiles with a numeric command line element do not raise errors.

3.11.20 0.6.4

- Fixed exception on stopping a container configuration when the container does not exist.

3.11.21 0.6.3

- Improved fixed behavior when merging container maps and embedded container configurations. Can also be used for creating copies.
- Added `stop_timeout` argument to `remove_all_containers`.
- Fixed transfer of configuration variables into client instance.

3.11.22 0.6.2

- Added `stop_signal` for customizing the signal that is used for shutting down or restarting containers.
- Minor changes in docs and log messages.
- Fixed image cache update with multiple tags.
- Bugfix in Dockerfile module.

3.11.23 0.6.1

- Many more Python 3 fixes (PR #10).
- Cleaned up logging; only using default levels.
- Port bindings are passed as lists to the API, allowing container ports to be published to multiple host ports and interfaces.

3.11.24 0.6.0

- Added `exec_commands` to start additional commands (e.g. scripts) along with the container.
- Container links are now passed as lists to the API, so that the same container can be linked with multiple aliases.
- Various compatibility fixes with Python 3 (PR #9).
- Bugfixes on container restart and configuration merge.

3.11.25 0.5.3

- Bugfixes for network mode and volume check of inherited configurations.
- Fixed deprecation warnings from `docker-py`.
- Added option to prepare attached volumes with local commands instead of temporary containers, for clients that support it.

3.11.26 0.5.2

- Added network modes and their dependencies. Attached volumes are no longer enabled for networking.
- Added per-container stop timeout. Also applies to restart.

3.11.27 0.5.1

- Adjusted volume path inspection to use `Mounts` on newer Docker API versions. Fixes issues with the update policy.

3.11.28 0.5.0

- Implemented HostConfig during container creation, which is preferred over passing arguments during start since API v1.15. For older API versions, start keyword arguments will be used.
- Added configuration inheritance and abstract configurations.
- Changed log functions to better fit Python logging.
- Minor fixes in merge functions.
- Bug fix in tag / repository partitioning (PR #7).

3.11.29 0.4.1

- Added automated container start, log, and removal for scripts or single commands.
- Added separate exception type for map integrity check failures.
- Aliases for host volumes are now optional.
- Minor bugfixes in late value resolution, container cleanup, and input conversion.

3.11.30 0.4.0

- Added check for changes in environment, command, and network settings in update policy.
- Added optional pull before new container creation.
- Revised dependency resolution for avoiding duplicate actions and detecting circular dependencies more reliably.
- Fix for handling missing container names in cleanup method.
- Allow for merging empty dictionary keys.

3.11.31 0.3.3

- Fix for missing container names and tags.
- Exclude default client name from host name.

3.11.32 0.3.2

- Fixed error handling in build (issue #6).
- New `command_workdir` for setting the working directory in DockerFiles.
- Enhanced file adding functions in DockerFile to return build context paths.
- Fixed volume consistency check in update policy.
- Additional minor updates.

3.11.33 0.3.1

- Extended late value resolution to custom types.
- Various bugfixes (e.g. PR #5).

3.11.34 0.3.0

- Possibility to use 'lazy' values in various settings (e.g. port bindings, volume aliases, host volumes, and user).
- Consider read-only option for inherited volumes in `uses` property.
- Further update policy fixes.
- Python 3 compatibility fixes (PR #4).

3.11.35 0.2.2

- Added convenience imports in `api` module.

3.11.36 0.2.1

- Added host and domain name setting.
- Improved update requirement detection.
- Fixed restart policy.

3.11.37 0.2.0

- Moved container handling logic to policy classes.
- Better support for multiple maps and multiple clients.
- Added `startup`, `shutdown`, and `update` actions, referring to variable policy implementations.
- Added `persistent` flag to container configurations to differentiate during cleanup processes.
- Added methods for merging container maps and configurations.
- It is no longer required to use the wrapped client `DockerClientWrapper`.
- More flexible logging.

3.11.38 0.1.4

- Minor fix in `DockerFile` creation.

3.11.39 0.1.3

- Only setup fix, no functional changes.

3.11.40 0.1.2

- Various bugfixes related to repository prefix, shortcuts, users.

3.11.41 0.1.1

- Added YAML import.
- Added default host root path and repository prefix.
- Added Docker registry actions to wrapper.
- Fixed issues related to starting containers.

3.11.42 0.1.0

Initial release.

CHAPTER 4

Status

Docker-Map 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 5

Indices and tables

- genindex
- modindex
- search