LXDock Documentation

Release 0.2.0

Virgil Dupras, Morgan Aubert

Apr 04, 2017

Contents

CHAPTER 1

Getting started

Requirements

- Python 3.4+
- LXD 2.0+
- `getfacl/setfacl` if you plan to use shared folders
- any provisioning tool you wish to use with LXDock

Building LXDock on Linux

LXDock should build very easily on Linux provided you have LXD available on your system.

Prerequisite: install LXD

You may want to skip this section if you already have a working installation of LXD on your system.

For Debian and Ubuntu, the following command will ensure that LXD is installed:

```
$ sudo apt-get install lxd
```

Note: If you're using an old version of Ubuntu you should first add the LXD's apt repository and install the `lxd` package as follows:

```
$ sudo add-apt-repository -y ppa:ubuntu-lxc/lxd-stable
$ sudo apt-get update
$ sudo apt-get install lxd
```

You should now be able to configure your LXD installation using:

```
$ newgrp lxd  # ensure your current user can use LXD
$ sudo lxd init
```

Note: The `lxd init` command will ask you to choose the settings to apply to your LXD installation in an interactive way (storage backend, network configuration, etc). But if you just want to go fast you can try the following commands (note that this will only work for **LXD 2.3+**):

```
$ newgrp lxd
$ sudo lxd init --auto
$ lxc network create lxdbr0 ipv6.address=none ipv4.address=10.0.3.1/24 ipv4.nat=true
$ lxc network attach-profile lxdbr0 default eth0
```

You can now check if your LXD installation is working using:

```
$ lxc launch ubuntu: first-machine && lxc exec first-machine bash
```

Note: You can use `lxc stop first-machine` to stop the previously created container.

Install LXDock

You should now be able to install LXDock using:

```
$ pip3 install lxdock
```

Note: Don't have `pip3` installed on your system? Most distros have a specific package for it, it's only a matter of installing it. For example, on Debian and Ubuntu, it's `python3-pip`. Otherwise, Stackoverflow can help you.

Command line completion

LXDock can provide completion for commands, options and container names.

Bash

If you use Bash, you have to make sure that bash completion is installed (which should be the case for most Linux installations). In order to get completion for LXDock, you should place the `contrib/completion/bash/lxdock` file at `/etc/bash.completion.d/lxdock` (or at any other place where your distribution keeps completion files):

```
$ sudo cp contrib/completion/bash/lxdock /etc/bash.completion.d/lxdock
```

Make sure to restart your shell before trying to use LXDock's bash completion.

ZSH

To add zsh completion for LXDock, place the `contrib/completion/zsh/_lxdock` file at `/usr/share/zsh/vendor-completions/_lxdock` (or another folder in `$fpath`):

```
$ sudo cp contrib/completion/zsh/_lxdock /usr/share/zsh/vendor-completions/_lxdock
```

Make sure to restart your shell before trying to use LXDock's zsh completion.

Your first LXDock file

Create a file called `.lxdock.yml` (or `lxdock.yml`) in your project directory and paste the following:

```
name: myproject

containers:
  - name: test01
    image: ubuntu/xenial

  - name: test02
    image: archlinux
```

This LXDock file defines a project (`myproject`) and two containers, `test01` and `test02`. These containers will be constructed using respectively the `ubuntu/xenial` and the `archlinux` images (which will be pulled from an image server - https://images.linuxcontainers.org by default).

Now from your project directory, start up your containers using the following command:

```
$ lxdock up
Bringing container "test01" up
Bringing container "test02" up
==> test01: Unable to find container "test01" for directory "[PATH_TO_YOUR_PROJECT]"
==> test01: Creating new container "myproject-test01-11943450" from image ubuntu/
↪xenial
==> test01: Starting container "test01"...
==> test01: No IP yet, waiting 10 seconds...
==> test01: Container "test01" is up! IP: [CONTAINER_IP]
==> test01: Doing bare bone setup on the machine...
==> test01: Adding ssh-rsa [SSH_KEY] to machine's authorized keys
==> test01: Provisioning container "test01"...
==> test02: Unable to find container "test02" for directory "[PATH_TO_YOUR_PROJECT]"
==> test02: Creating new container "myproject-test02-11943450" from image archlinux
==> test02: Starting container "test02"...
==> test02: No IP yet, waiting 10 seconds...
==> test02: Container "test02" is up! IP: [CONTAINER_IP]
==> test02: Doing bare bone setup on the machine...
==> test02: Adding ssh-rsa [SSH_KEY] to machine's authorized keys
==> test02: Provisioning container "test02"...
```

Congrats! You're in!

Problems?

If you're having problems trying to run your container, try running them in *privileged* mode. Many older distributions have an init system that doesn't work well with unprivileged containers (*debian/jessie* notably). Some host-side problems can also be worked around by running privileged containers.

If you received a permission denied error running the lxc network commands below:

```
$ lxc network create lxdbr0 ipv6.address=none ipv4.address=10.0.3.1/24 ipv4.nat=true
$ lxc network attach-profile lxdbr0 default eth0
```

Run these commands below and then run the lxc network commands again. You should now be able to proceed with the remaining instructions.

```
$ sudo systemctl stop lxd.socket
$ sudo systemctl start lxd.socket
```

CHAPTER 2

Usage

Here is a list of simple guides targeting specific use cases that are mostly to be encountered when using LXDock.

Multiple containers

You can define multiple containers in your LXDock file. All you have to do is to use the `containers` section and define your containers below it.

```
image: ubuntu/xenial
mode: pull

containers:
  - name: web
    hostnames:
      - myproject.local

  - name: ci
    image: debian/jessie
    privileged: true
    hostnames:
      - ci.local
```

If you define some global values (eg. `images`, `mode` or `provision`) outside of the scope of the `containers` block, these values will be used when creating each container unless you re-define them in the container's configuration scope.

Provisioning

LXDock supports many provisioning tools in order to allow you to easily provision containers created using LXD. Using provisioning tools such as Ansible with LXDock will allow you to alter the configuration, install software, deploy applications and more on the containers. Using the built-in provisioning capabilities of LXDock will allow you to run these provisioning operations as part of the `lxdock up` wokflow. To be more precise, the provisioning tools associated with your LXDock configuration are executed in the following situations:

- when you run `lxdock up` the first time; that is when the container does not exist yet
- when you run `lxdock provision`. Note that you can run this command as many time as you want

Currently, LXDock provides a built-in support for the following provisioning tools:

- Ansible
- *Your favorite provisioning tool is not listed here?!!* Feel free to contribute!

The provisioning tools you choose to use can be configured in your LXDock file using the `provisioning` option. For example, we could choose to provision our containers using an Ansible playbook as follows:

```
name: myproject
image: ubuntu/xenial

provisioning:
  - type: ansible
    playbook: deploy/site.yml
```

Note that you can use *many* provisioning tools. The order in which provisioning tools are defined in your LXdock file defines the order in which they are executed.

Note: Please refer to *Provisioners* to see the full list of supported provisioners.

Shared folders

A common need when using a tool such as LXDock is to make some folders on your system available to your containers. LXC/LXD provides a feature called “lxc mounts” - LXDock uses it internally in order to provide support for “shared folders”.

You can use the `shares` option in order to define which folders should be made available to your containers. For example:

```
name: myproject
image: ubuntu/xenial

shares:
  - source: /path/to/my/workspace/project/
    dest: /myshare
```

Of course you can associate many shared folders with your containers. In the previous example, the content of the `/path/to/my/workspace/project/` on the host will be made available to the container under the `/myshare` folder.

The problem with shared folder permissions

Shared folders in LXDock use lxc mounts. This is simple and fast, but there are problems with permissions: shared folders means shared permissions. Changing permissions in the container means changing them in the host as well, and vice versa. That leaves us with a problem that is tricky to solve gracefully. Things become more complicated when our workflow has our container create files in that shared folder. What permissions do we give these files?

LXDock tries to answer this by using ACLs. To ensure that files created by the container are accessible to you back on the host (and vice versa), every new share has a default ACL giving the current user full access to the source folder. An ACL is also added for the root user of the container in order to allow him to access the shared folders on the guest side with read/write permissions.

You should note that users created by your provisioning tools (eg. using Ansible) won't be able to access your shares on the guest side. This is because LXDock has no knowledge of the users who should have access to your shares. Moreover, your users/groups, when the container is initially created, don't exist yet! That is why it does nothing. What is suggested is that you take care of it in your own provisioning by setting up some ACLs. You can also make use of the `users` option in order to force LXDock to create some users. The users created this way will be handled by LXDock and will have read/write access to the shared folders:

```
name: myproject
image: ubuntu/xenial

shares:
  - source: /path/to/my/workspace/project/
    dest: /myshare

users:
  - name: test01
  - name: test02
    home: /opt/test02
```

CHAPTER 3

Command-line reference

Most of your interaction with LXDock will be done using the `lxdock` command. This command provides many subcommands: `up`, `halt`, `destroy`, etc. These subcommands are described in the following pages but you can easily get help using the `help` subcommand. `lxdock help` will display help information for the `lxdock` command while `lxdock help [subcommand]` will show the help for a specifc subcommand. For example:

```
$ lxdock help up
usage: lxdock up [-h] [name [name ...]]

Create, start and provision all the containers of the project according to
your LXDock file. If container names are specified, only the related containers
are created, started and provisioned.

positional arguments:
  name        Container name.

optional arguments:
  -h, --help  show this help message and exit
```

lxdock config

Command: `lxdock config`

This command can be used to validate and print the LXDock config file of the project.

Options

- `--containers` - prints only container names, one per line

Examples

```
$ lxdock config                   # prints project's LXDock file
$ lxdock config --containers      # prints project's container names
```

lxdock destroy

Command: `lxdock destroy [name [name ...]]`

This command can be used to destroy containers. If the containers to be destroyed are still running they will first be stopped.

By default this command will try to destroy all the containers of the current project but you can limit this operation to some specific containers by specifying their names. Keep in mind that a confirmation will be prompted to the user when using the *destroy* command.

Options

- `[name [name ...]]` - zero, one or more container names
- `--force` or `-f` - this option allows to destroy containers without confirmation

Examples

```
$ lxdock destroy                 # destroys all the containers of the project
$ lxdock destroy mycontainer     # destroys the "mycontainer" container
$ lxdock destroy web ci          # destroys the "web" and "ci" containers
$ lxdock destroy --force web     # destroys the "web" container without confirmation
```

lxdock halt

Command: `lxdock halt [name [name ...]]`

This command can be used to halt running containers.

By default this command will try to halt all the containers of the current project but you can limit this operation to some specific containers by specifying their names.

Options

- `[name [name ...]]` - zero, one or more container names

Examples

```
$ lxdock halt                 # halts all the containers of the project
$ lxdock halt mycontainer     # halts the "mycontainer" container
$ lxdock halt web ci          # halts the "web" and "ci" containers
```

lxdock help

Command: `lxdock help [subcommand]`

This command can be used to show help information.

By default this command will show the global help information for the `lxdock` cli but you can also get help information for a specific subcommand.

Options

- `[subcommand]` - a subcommand name (eg. `up`, `halt`, ...)

Examples

```
$ lxdock help                  # shows the global help information
$ lxdock help destroy          # shows help information for the "destroy" subcommand
```

lxdock init

Command: `lxdock init`

This command can be used to generate a LXDock file containing highlights regarding some useful options.

Options

- `--image` - this option allows to use a specific container image in the generated configuration
- `--project` - this option allows to define the name of the project that will appear in the LXDock file
- `--force` or `-f` - this option allows to overwrite an exsting LXDock file if any

Examples

```
$ lxdock init                         # generates a basic LXDock file
$ lxdock init --image debian/jessie    # generates a LXDock file defining a debian/
↪jessie container
$ lxdock init --project myproject      # generates a basic LXDock file defining a
↪"myproject" project
$ lxdock init --force                  # overwrite an existing LXDock file if␣
↪applicable
```

lxdock provision

Command: `lxdock provision [name [name ...]]`

This command can be used to provision your containers.

By default it will install bare bones packages (openssh, python) into your container if the underlying distribution is supported by LXDock. That said, the `provision` command can also trigger the execution of provisioning tools that you could've configured in your LXDock file (using the `provisioning` block).

Options

- `[name [name ...]]` - zero, one or more container names

Examples

```
$ lxdock provision                 # provisions all the containers of the project
$ lxdock provision mycontainer     # provisions the "mycontainer" container
$ lxdock provision web ci          # provisions the "web" and "ci" containers
```

lxdock shell

Command: `lxdock shell [arguments] [name]`

This command can be used to open an interactive shell inside one of your containers.

By default, that shell logins as `root` unless your LXDock config specifies another user in its `shell:` option. In all cases, the `--user` command line overrides everything.

Options

- `[name]` - a container name
- `-u, --user <username>` - user to login as

Examples

```
$ lxdock shell mycontainer         # opens a shell into the "mycontainer" container
$ lxdock shell -u root             # opens a shell as root, regardless of our config
```

lxdock status

Command: `lxdock status [name [name ...]]`

This command can be used to show the statuses of the containers of your project.

By default this command will display the statuses of all the containers of your project but you can limit this operation to some specific containers by specifying their names. The statuses that are returned by this command can be `not-created`, `stopped` or `running`.

Options

- `[name [name ...]]` - zero, one or more container names

Examples

```
$ lxdock status               # shows the statuses of all the containers of the␣
↪project
$ lxdock status mycontainer   # shows the status of the "mycontainer" container
$ lxdock status web ci        # shows the statuses of the "web" and "ci" containers
```

lxdock up

Command: `lxdock up [name [name ...]]`

This command can be used to start the containers of your project.

By default this command will try to start all the containers of your project but you can limit this operation to some specific containers by specifying their names. It should be noted that containers will be created (and provisioned) if they don't exist yet.

Options

- `[name [name ...]]` - zero, one or more container names

Examples

```
$ lxdock up                # starts the containers of the project
$ lxdock up mycontainer    # starts the "mycontainer" container
$ lxdock up web ci         # starts the "web" and "ci" containers
```

CHAPTER 4

LXDock file reference

LXDock files allow you to defines which containers should be created for your projects. LXDock files are YML files and should define basic information allowing LXDock to properly create your containers (eg. container names, images, ...). By default LXDock will try to use a file located at `./.lxdock.yml`.

Note: LXDock supports the following names for LXDock files: `.lxdock.yml`, `lxdock.yml`, `.lxdock.yaml` and `lxdock.yaml`.

A container definition contains parameters that will be used when creating each container of a specific project. It should be noted that most of the options that you can define in your LXDock file can be applied "globally" or in the context of a specific container. For example you can define a global `image` option telling to use the `ubuntu/xenial` for all your containers and decide to use the `debian/jessie` image for a specific container:

```
name: myproject
image: ubuntu/xenial

containers:
  - name: test01
  - name: test02
  - name: test03
    image: debian/jessie
```

This section contains a list of all configuration options supported by LXDock files.

containers

The `containers` block allows you to define the containers of your project. It should be a list of containers, as follows:

```
name: myproject
image: ubuntu/xenial
```

```
containers:
  - name: test01
  - name: test02
```

environment

A mapping of environment variables to override in the container when executing commands. This will affect `lxdock shell` and `lxdock provision` operations.

```
name: myproject
image: ubuntu/xenial

environment:
  LC_ALL: en_US.utf8
```

hostnames

The `hostnames` option allows you to define which hostnames should be configured for your containers. These hostnames will be added to your `/etc/hosts` file, thus allowing you to easily access your applications or services.

```
name: myproject
image: ubuntu/xenial

containers:
  - name: test01
    hostnames:
      - myapp.local
      - myapp.test
  - name: test02
```

image

The `image` option should contain the alias of the image you want to use to build your containers. LXDock will try to pull images from the default LXD's image server. So you can get a list of supported aliases by visiting https://images.linuxcontainers.org/ or by listing the aliases of the "images:" default remote:

```
$ lxc image alias list images:
```

There are many scenarios to consider when you have to choose the value of the `image` option. If you choose to set your `image` option to `ubuntu/xenial` this means that the container will use the Ubuntu's Xenial version with the same architecture as your host machine (amd64 in most cases). It should be noted that the `image` value can also contain a container alias that includes the targetted architecture (eg. `debian/jessie/amd64` or `ubuntu/xenial/armhf`).

Here is an example:

```
name: myproject
image: ubuntu/xenial
```

You should note that you can also use "local" container aliases. This is not the most common scenario but you can manage your own image aliases and decide to use them with LXDock. You'll need to use the `mode: local` option if you decide to do this (the default `mode` is `pull`). For example you could create an image associated with the `old-ubuntu` alias using:

```
$ lxc image copy ubuntu:12.04 local: --alias old-ubuntu
```

And then use it in your LXDock file as follows:

```
name: myproject
image: old-ubuntu
mode: local
```

mode

The `mode` option allows you to specify which mode to use in order to retrieve the images that will be used to build your containers. Two values are allowed here: `pull` (which is the default mode for LXDock) and `local`. In `pull` mode container images will be pulled from an image server (https://images.linuxcontainers.org/ by default). The `local` mode allows you to use local container images (it can be useful if you decide to manage your own image aliases and want to use them with LXDock).

name

This option can define the name of your project or the name of a container. In either cases, the `name` option is mandatory.

```
name: myproject
image: ubuntu/xenial

containers:
  - name: container01
  - name: container01
```

privileged

You should use the `privileged` option if you want to created privileged containers. Containers created by LXDock are unprivileged by default. Such containers are safe by design because the root user in the containers doesn't map to the host's root user: it maps to an unprivileged user *outside* the container.

Here is an example on how to set up a privileged container in your LXDock file:

```
name: myproject
image: ubuntu/xenial

containers:
  - name: web
    privileged: yes
```

Note: Please refer to *Glossary* for more details on these notions.

protocol

The `protocol` option defines which protocol to use when creating containers. By default LXDock uses the `simplestreams` protocol (as the `lxc` command do) but you can change this to use the `lxd` protocol if you want. The `simplestreams` protocol is an image server description format, using JSON to describe a list of images and allowing to get image information and import images. The `lxd` protocol refers to the REST API that is used between LXD clients and LXD daemons.

provisioning

The `provisioning` option allows you to define how to provision your containers as part of the `lxdock up` workflow. This provisioning can also be executed when running `lxdock provision`.

The `provisioning` option should define a list of provisioning tools to execute. For example, it can be an Ansible playbook to run:

```
name: myproject
image: ubuntu/xenial

provisioning:
  - type: ansible
    playbook: deploy/site.yml
```

Note: Please refer to *Provisioners* to see the full list of supported provisioners.

server

You can use this option to define which image server should be used to retrieve container images. By default we are using https://images.linuxcontainers.org/.

shares

The `shares` option lets you define which folders on your host should be made available to your containers (internally this feature uses lxc mounts). The `shares` option should define a list of shared items. Each shared item should define a `source` (a path on your host system) and a `dest` (a destination path on your container filesystem). For example:

```
name: myproject
image: ubuntu/xenial

shares:
  - source: /path/to/my/workspace/project/
    dest: /myshare
```

shell

The `shell` option alters the behavior of the `lxdock shell` command. It's a map of these sub-options:

- `user`: Default user to open the shell under.
- `home`: Path to open the shell under.

```
name: myproject
image: ubuntu/xenial

shell:
  user: myuser
  home: /opt/myproject
```

users

The `users` option allows you to define users that should be created by LXDock after creating a container. This can be useful because the users created this way will automatically have read/write permissions on shared folders. The `users` option should contain a list of users; each with a `name` and optionally a custom `home` directory or custom `password`.

Passwords are encrypted using crypt(3) as explained in the useradd manpage, see `man useradd` for more information:

```
name: myproject
image: ubuntu/xenial

users:
  - name: test01
  - name: test02
    home: /opt/test02
  - name: test03
    password: $6$cGzZBkDjOhGW
↪$6C9wwqQteFEY4lQ6ZJBggE568SLSS7bIMKexwOD39mJQrJcZ5vIKJVIfwsKOZajhbPw0.
↪Zqd0jU2NDLAnp9J/1
```

CHAPTER 5

Provisioners

LXDock provides support for common provisioning operations. Provisioning operations can be easily defined in your LXDock file using the `provisioning` option:

```
name: myproject
image: ubuntu/xenial

provisioning:
  - type: ansible
    playbook: deploy/site.yml
```

Warning: When using provisioners you should keep in mind that some of them can execute local actions on the host. For example Ansible playbooks can trigger local actions that will be run on your system. Other provisioners (like the shell provisioner) can define commands to be runned on the host side or in provileged containers. **You have to** trust the projects that use these provisioning tools before running LXDock!

Documentation sections for the supported provisioning tools or methods are listed here.

Ansible

LXDock provides built-in support for Ansible provisioning.

Requirements

Ansible v2.0+

Usage

Just append an `ansible` provisioning operation to your LXDock file as follows:

```
name: myproject
image: ubuntu/xenial

provisioning:
  - type: ansible
    playbook: deploy/site.yml
```

Required options

playbook

The `playbook` option allows you to define the path to your Ansible playbook you want to run when your containers are provisioned.

Optional options

ask_vault_pass

You can use this option to force the use of the `--ask-vault-pass` flag when the `ansible-playbook` command is executed during the provisioning workflow. Here is an example:

```
[...]
provisioning:
  - type: ansible
    playbook: deploy/site.yml
    ask_vault_pass: yes
```

vault_password_file

You can use this option to specify the path toward the vault password file you want to use when the `ansible-playbook` command is executed. Here is an example:

```
[...]
provisioning:
  - type: ansible
    playbook: deploy/site.yml
    vault_password_file: secrets/vaultpass
```

Shell

The shell provisioner allows you to execute commands on the guest side or the host side in order to provision your containers.

Usage

Just append a `shell` provisioning operation to your LXDock file as follows:

```
name: myproject
image: ubuntu/xenial

provisioning:
  - type: shell
    inline: echo "Hello, World!"
```

Note: Keep in mind that the shell provisioner will use the LXD's `exec` method in order to run your commands on containers (the same method used by the `lxc exec` command). This means that common shell patterns (like file redirects, `|`, `>`, `<`, ...) won't work because the `exec` method doesn't use a shell (so the kernel will not be able to understand these shell patterns). The only way to overcome this is to put things like `sh -c 'ls -l > /tmp/test'` in your `inline` options.

Required options

inline

The `inline` option allows you to specify a shell command that should be executed on the guest side or on the host. Note that the `inline` option and the `script` option are mutually exclusive.

```
[...]
provisioning:
  - type: shell
    inline: echo "Hello, World!"
```

script

The `script` option lets you define the path to an existing script that should be executed on the guest side or on the host. Note that the `script` option and the `inline` option are mutually exclusive.

```
[...]
provisioning:
  - type: shell
    script: path/to/my/script.sh
```

Optional options

side

Use the `side` option if you want to define that the shell commands/scripts should be executed on the host side. The default value for this option is `guest`. Here is an example:

```
[...]
provisioning:
  - type: shell
    side: host
    inline: echo "Hello, World!"
```

CHAPTER 6

Glossary

This is a comprehensive list of the terms used when discussing the functionalities and the configuration options of LXDock.

Container Or *Linux containers*. Whenever we use the term “container”, we are referring to LXD containers. LXD focuses on system containers / infrastructure containers and thus provides an elegant solution to the problem of how to reliably run software in multiple computing environments (eg. for development or tests execution).

Image An image (or container image) is necessary to build a container. Basically container images embed a snapshot of a full filesystem and some configuration-related tools. All containers are built from “local” images; but images can also be pulled from a remote image server (the default LXD’s image server is at https://images.linuxcontainers.org/). This a good option because users don’t have to manage their own images but they have to trust the image server they are using!

LXC LXC stands for “Linux containers”. It is a technology that allows to virtualize software (which can be an entire operating system) at the operating system level, within the Linux kernel.

LXD LXD can be seen as an extension of LXC. It’s a container system that makes use of LXC. It provides many tools built around LXC such as a REST API to interact with your containers, an intuitive command line tool, a container image system, ...

Privileged container Privileged containers are containers where the root user (in the container) is mapped to the host’s root user. This is not really “root-safe” and could lead to potential security flawns. That said it should be noted that privileged containers come with some protection mechanisms in order to protect the host. You can refer to LXC’s documentation for more details on this topic.

Unprivileged container Unprivileged containers are containers where the root user (in the container) is mapped to an unprivileged container on the host. So the user that corresponds to the container’s root user only has advanced rights and permissions on the resources related to the container it is associated to.

CHAPTER 7

Contributing to LXDock

Here are some simple rules & tips to help you contribute to LXDock. You can contribute in many ways!

Contributing code

The preferred way to contribute to LXDock is to submit pull requests to the project's Github repository. Here are some general tips regarding pull requests.

Warning: Keep in mind that you should propose new features on the project's issue tracker before starting working on your ideas!

Development environment

You should first fork the LXDock's repository and make sure that LXD is properly installed on your system. Then you can get a working copy of the project using the following commands (eg. using Python 3.6):

```
$ git clone git@github.com:<username>/lxdock.git
$ cd lxdock
$ python3.6 -m venv ./env && . ./env/bin/activate
$ make install
```

Coding style

Please make sure that your code is compliant with the PEP8 style guide. You can ignore the "Maximum Line Length" requirement but the length of your lines should not exceed 100 characters. Remember that your code will be checked using flake8 and isort. You can use the following commands to perform these validations:

```
$ make lint
$ make isort
```

Or:

```
$ tox -e lint
$ tox -e isort
```

Tests

You should not submit pull requests without providing tests. LXDock relies on pytest: py.test is used instead of unittest for its test runner but also for its syntax. So you should write your tests using pytest instead of unittest and you should not use the built-in `TestCase`.

You can run the whole test suite using the following command:

```
$ py.test
```

Code coverage should not decrease with pull requests! You can easily get the code coverage of the project using the following command:

```
$ make coverage
```

Using the issue tracker

You should use the project's issue tracker if you've found a bug or if you want to propose a new feature. Don't forget to include as many details as possible in your tickets (eg. tracebacks if this is appropriate).

CHAPTER 8

Release notes

Here are listed the release notes for each version of LXDock.

LXDock 0.2

LXDock 0.2 release notes (2017-04-04)

Requirements and compatibility

Python 3.4, 3.5 and 3.6. LXD 2.0+.

New features

- Add support for ZSH completion (#39)
- Add the possibility to use the `lxdock` command from subfolders (#45)
- Add support for ansible options related to ansible-vault (`ask_vault_pass`, `vault_password_file`) (#54)
- Add support for shell provisioning (#56)
- Add a `password` option to set the password of users created using the `users` option (#60)
- Add support for environment variables mapping (`environment` option) (#50)

Improvements

- Improve documentation (#49, #43)
- Fix integration tests flakiness

LXDock 0.1

LXDock 0.1.1 release notes (2017-03-09)

Requirements and compatibility

Python 3.4, 3.5 and 3.6. LXD 2.0+.

Fixes

- Fixed wrong container names in Bash completion (#38)

LXDock 0.1 release notes (2017-03-09)

Requirements and compatibility

Python 3.4, 3.5 and 3.6. LXD 2.0+.

New features

This is the initial release of LXDock!

CHAPTER 9

Thanks

We would like to thank Savoir-faire Linux for allowing us to work on this side project! Developers at Savoir-faire Linux use LXDock on a daily basis to manage local infrastructure containers related to DevOps projects.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Index