LXDock Documentation

Release 0.5.0.dev

Virgil Dupras, Morgan Aubert

Jul 13, 2018

Contents

CHAPTER 1

Getting started

1.1 Requirements

- Python 3.4+
- LXD 2.0+
- any provisioning tool you wish to use with LXDock

1.2 Building LXDock on Linux

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

1.2.1 Prerequisite: install LXD

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

LXD is available in the repository for Debian or Ubuntu 16.04 and higher:

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

You can now also install LXD from Snap which works on Ubuntu 14.04 and higher. Since the LXD PPA has been deprecated, this is now the easiest way to get the latest version of LXD on Ubuntu.

Note: If you have already installed and configured LXD from apt earlier and want to upgrade to the Snap version, you may need to purge LXD packages first and reboot for the old network bridge to be removed. You can migrate existing containers using `lxd.migrate` or just start fresh.

```
$ sudo apt-get purge lxd lxd-client
```

To install LXD from a Snap instead of apt:

```
$ sudo apt-get install snapd
$ sudo snap install lxd
$ sudo snap start lxd
```

For Fedora, LXD is available through an experimental COPR repository. Unfortunately SELinux is not yet supported, therefore make sure it is disabled or set to permissive. Then run:

```
$ dnf copr enable ganto/lxd
$ dnf install lxd lxd-tools
```

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.

1.2.2 Prepare host for shared folders

LXDock uses `raw.idmap` for shared folders to so that files on the share that are owned by the host user appear to be owned by the container user inside the container, even if new files are created inside the container.

To use shares, the following needs to be run once to prepare the host, then LXD needs to be restarted.

```
$ printf "lxd:$(id -u):1\nroot:$(id -u):1\n" | sudo tee -a /etc/subuid
$ printf "lxd:$(id -g):1\nroot:$(id -g):1\n" | sudo tee -a /etc/subgid
```

To restart LXD use `sudo snap restart lxd` or `sudo service restart lxd` or equivalent for your system.

1.2.3 Install LXDock

You should now be able to install LXDock using:

```
$ pip3 install lxdock
```

Note: It is good practice to install lxdock in a virtualenv rather than installing it globally as root, but make sure you always use a python3 virtualenv. To use lxdock from any location without having to activate this virtualenv, you can create a symlink from the lxdock executable in the virtualenv to `/usr/bin/lxdock` or `/usr/local/bin/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.

1.3 Command line completion

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

1.3.1 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 curl -L https://raw.githubusercontent.com/lxdock/lxdock/$(lxdock --version |␣
↪cut -d ' ' -f 2)/contrib/completion/bash/lxdock -o /etc/bash_completion.d/lxdock
```

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

1.3.2 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 curl -L https://raw.githubusercontent.com/lxdock/lxdock/$(lxdock --version |␣
↪cut -d ' ' -f 2)/contrib/completion/zsh/_lxdock -o /usr/share/zsh/vendor-
↪completions/_lxdock
```

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

1.4 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
```

(continues on next page)

(continued from previous page)

```
    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!

1.5 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.

2.1 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.

2.2 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.

2.3 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.

2.3.1 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
```

2.3.2 Disabling ACL support on shares

By default ACLs will be turned on for all shares, however it is also possible to disable this functionality on a per-share basis. One reason you might want to do this, is when you are using privileged containers and ensuring the container user matches the uid and gid of the host system. This allows a share to be mapped without the use of ACLs, however the user should be aware of the security implications of making shares world-writable. This may be acceptable for development only containers for example.

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

shares:
  - source: .
    dest: /myshare
    set_host_acl: false

users:
  - name: test01
```

In this example, the Ansible provisioner can be used to change the uid and gid of the test01 user after it has been created by LXDock. How to implement this is up to the user, as LXDock does not provide a uid and gid option when creating users.

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
```

3.1 lxdock config

Command: `lxdock config`

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

3.1.1 Options

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

3.1.2 Examples

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

3.2 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.

3.2.1 Options

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

3.2.2 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
```

3.3 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.

3.3.1 Options

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

3.3.2 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
```

3.4 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.

3.4.1 Options

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

3.4.2 Examples

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

3.5 lxdock init

Command: `lxdock init`

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

3.5.1 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

3.5.2 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
```

3.6 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).

3.6.1 Options

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

3.6.2 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
```

3.7 lxdock shell

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

This command can be used to open an interactive shell inside one of your containers. If `-c` is specified, execute the command line instead of opening the shell.

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.

3.7.1 Options

- `[name]` - a container name
- `-u, --user <username>` - user to login as
- `-c, --command <command_line>` - command to be executed in the shell

3.7.2 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 shell mycontainer -c echo HELLO      # executes "echo HELLO" in "mycontainer"
$ lxdock shell mycontainer -c echo '$PATH'    # executes "echo '$PATH'" in
↪"mycontainer"
```

For the last example, you will see "$PATH" as-is. It is not evaluated as a variable.

3.8 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`.

3.8.1 Options

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

3.8.2 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
```

3.9 lxdock up

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

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.

3.9.1 Options

- `[name [name ...]]` - zero, one or more container names
- `--provision` - this option allows to force containers to be provisioned
- `--no-provision` - this option allows to disable container provisioning

3.9.2 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
$ lxdock up --provision     # starts the containers of the project and provision them␣
↪(even if they were already created)
$ lxdock up --no-provision  # starts the containers of the project but disable the␣
↪provisioning step
```

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.

4.1 Supported options

4.1.1 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
```

4.1.2 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
```

4.1.3 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
```

4.1.4 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
```

4.1.5 lxc_config

If your container needs custom configuration settings, you can use `lxc_config` to pass any arbitrary key-value pairs that you want to be assigned to the container at startup. `lxc_config` must be a dictionary that with an arbitrary number of key-values.

```
name: myproject
image: ubuntu/xenial
lxc_config:
  global_key: global_value

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

4.1.6 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).

4.1.7 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
```

4.1.8 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.

4.1.9 profiles

The `profiles` option defines the configuration profiles to use when creating containers. It should contain a list of profile names that exist in `lxc profile list`. Please use `lxc profile` command to manage profiles as they are system-wide.

```
name: myproject
image: ubuntu/xenial
profiles:
  - default
  - docker

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

4.1.10 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.

4.1.11 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
```

Unlike normal configuration options, whether it is declared globally or specifically to a container has a special meaning for some provisioners. For example, a global `ansible` provisioner will only run **once** with an inventory including all containers. For some other provisioner, the global context doesn't make sense, so in these cases, these provisioner will be ran individually on each container.

Like with other configuration options, however, you can of course declare a `provisioning` section locally to target a specific container.

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

4.1.12 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/.

4.1.13 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
    set_host_acl: true
```

Note: The `set_host_acl` parameter is optional and defaults to true when left out, please refer to *Shared folders* for more information.

4.1.14 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
```

4.1.15 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
```

4.2 Variable substitution

LXDock files can contain variables. These variables are processed when your LXDock file is parsed by LXDock and can be defined in multiple places:

- in your host's environment variables
- in a `.env` file placed at the root of your project (that is, where your LXDock file is present)

Note that the above list represents the order of precedence that is used by LXDock internally. The last listed variables of this list will always win prioritization if variables are defined in multiple places.

4.2.1 Variables definitions

Suppose you want to create a user whose name corresponds to your current username on the host. You could create a LXDock file which looks like:

```
name: myproject
image: ubuntu/xenial

users:
  - name: ${USER}
```

When processing this file, LXDock will look for the `USER` environment variable (which can be defined either in your host's environment variables or in a `.env` file) and will substitute its variable in.

4.2.2 Where to put variables

As explained above, you can set your environment variables in multiple locations:

- in your host's environment variables
- in a `.env` file placed at the root of your project

For example you may want to set a specific environment variable before running LXDock:

```
$ export MY_VAR=test
$ lxdock up     # MY_VAR=test will be used by LXDock
```

You can also create a `.env` file at the root of your project. Such a file should only contain key/value pairs:

```
# A .env file!
MY_VAR = thisisatest
LXDOCK_TEST=42
```

4.2.3 Predefined variables

LXDock provides some default variables that you can use in your LXDock files. These variables are automatically set by LXDock when your configuration file is processed.

LXDOCK_YML_DIR

The `LXDOCK_YML_DIR` variable can be used to refer to the absolute path where your LXDock file lives.

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.

5.1 Ansible

LXDock provides built-in support for Ansible provisioning.

5.1.1 Requirements

Ansible v2.0+

5.1.2 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
```

5.1.3 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.

5.1.4 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
```

groups

Use this option to add groups to the inventory that is dynamically generated by the provisioner. This option is a hash of lists of container names. Example:

```
[...]
provisioning:
  - type: ansible
    playbook: deploy/site.yml
    groups:
      group1:
        - container1
```

(continues on next page)

(continued from previous page)

```
      - container2
    group2:
      - container2
```

lxd_transport

If this boolean option is set to `true`, we will use ansible's `lxd` transport instead of the `ssh` one, thus bypassing ssh entirely and using `lxc exec` directly.

It should be noted that while very cool-sounding, this transport method comes with a couple of drawbacks due to its incomplete support in Ansible. For example, `synchronize` actions don't work.

5.2 Puppet

LXDock provides built-in support for Puppet provisioning.

5.2.1 Distribution Support

Puppet provisioning requires the executable `puppet` in the guest container. For ArchLinux, Debian, Fedora and Ubuntu guests, LXDock tries to install the `puppet` package during `lxdock up`.

However, the automatic installation may fail, it may install an old version of Puppet, or you may also use another Linux distribution. In these cases, you may see an error message at the beginning of `lxdock provision` or during the provisioning (due to version mismatch). You will have to check out the proper documentation for your Linux distribution and Puppet version, and add a shell provisioner in LXDock file before the Puppet provisioner. Here is an example for CentOS 7:

```
name: centos-7
image: centos/7

provisioning:
  - type: shell
    inline: sh -c 'rpm -ivh https://yum.puppetlabs.com/puppetlabs-release-el-7.noarch.
↪rpm && yum -y install puppet'
  - type: puppet
    [...] # Puppet options
```

By default, LXDock runs `which puppet` to find the executable. You may override this behavior by providing `binary_path` option and LXDock will then find `puppet` executable under that path.

5.2.2 Usage

Add a `puppet` provisioning operation to your LXDock file as follows:

```
name: myproject
image: ubuntu/xenial

provisioning:
  - type: puppet
    manifests_path: manifests
```

(continues on next page)

(continued from previous page)

```
    manifest_file: site.pp
    module_path: modules
    hiera_config_path: hiera.yaml
    facter:
        role: app
        domain_name: app.example.com
    options: "--verbose --debug"
```

Puppet provisioning can be run in "manifest" mode or "environment" mode by setting `manifests_path` or `environment_path`, respectively. If `manifest_file` is not provided in "manifest" mode, it is given the default value `default.pp`. If `environment` is not provided in "environment" mode, it is given `production` as the default value.

If none of `manifests_path` and `environment_path` are given, LXDock assumes "manifest" mode and set `manifests_path` to `manifests`. During the validation of an LXDock file, the existence of `manifests_path/manifest_file` or `environment_path/environment` is checked.

5.2.3 Options

LXDock's Puppet provisioning is expected to reuse the files and configurations for a Vagrant project. This is still experimental, so if it doesn't work for your case, please feel free to create a GitHub issue!

Here are the options that LXDock has supported:

- `binary_path`
- `facter`
- `hiera_config_path`
- `manifest_file`
- `manifests_path`
- `module_path`
- `environment`
- `environment_path`
- `environment_variables`
- `options`: LXDock takes a single string of space-separated options, instead of an array of strings.

Please reference: Vagrant docs

5.3 Shell

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

5.3.1 Usage

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

```
name: myproject
image: ubuntu/xenial

provisioning:
  - type: shell
    inline: cd /tmp && echo "Here's the PATH" $$PATH >> test.txt
```

Note: The inline command is executed by `sh -c 'command_line'`. Keep in mind that dollar sign `$` means string interpolation in YAML and it is necessary to put `$$` to escape the dollar sign.

5.3.2 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
```

5.3.3 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 flaws. 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!

7.1 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!

7.1.1 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
```

Instead of setting up an environment directly on your development machine itself, you can also use the included Vagrantfile for creating a test environment:

```
$ vagrant up
$ vagrant ssh
$ make coverage
```

7.1.2 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
```

7.1.3 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 but in some cases we realise that isn't always possible. If this happen it should be discussed on a Github issue with the maintainers, so that there is a record of it.

You can easily get the code coverage of the project using the following command:

```
$ make coverage
```

7.2 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.

8.1 LXDock 0.4

8.1.1 LXDock 0.4.1 release notes (2017-10-02)

Requirements and compatibility

Python 3.4, 3.5 and 3.6. LXD 2.0+.

Fixes

- Fix packaging issue (#100)

8.1.2 LXDock 0.4 release notes (2017-10-02)

Requirements and compatibility

Python 3.4, 3.5 and 3.6. LXD 2.0+.

New features

- Add a `lxd_transport` option to the Ansible provisioner to make use of the ansible's `lxd` transport instead of the `ssh` one (#94)
- Add a `groups` option to the Ansible provisioner (#95)

8.2 LXDock 0.3

8.2.1 LXDock 0.3 release notes (2017-07-10)

Requirements and compatibility

Python 3.4, 3.5 and 3.6. LXD 2.0+.

New features

- Add support for passing custom LXC parameters for containers in LXDock files (#47)
- Add support for variable interpolation in LXDock files (#65, #75)
- Add support for shell patterns on the `inline` option of shell provisioner (#70)
- Add support for passing a command line with `lxdock shell` (#67)
- Add basic support for LXC profile in LXDock files (#77)
- Add a `set_host_acl` option to disable setting host ACL on a share (#79)
- Add support for provisioning-specific arguments to the `up` command (#84)
- Add experimental support for Puppet provisioning (#69)
- Add global provisioning capabilities (#90)

Minor changes

- Fixed wrong guest names and ssh setup in some Ansible provisioner (#71)
- Ensure the container is created when running provision or shell subcommands

8.3 LXDock 0.2

8.3.1 LXDock 0.2.1 release notes (2017-05-30)

Requirements and compatibility

Python 3.4, 3.5 and 3.6. LXD 2.0+.

Fixes

- Pin `requests` to avoid severe problems caused by new releases (#81)

8.3.2 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

8.4 LXDock 0.1

8.4.1 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)

8.4.2 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