LXD-Nomad Documentation

Release 0.2.0.dev

Virgil Dupras, Morgan Aubert

February 20, 2017

Contents

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 LXD-Nomad

Building nomad on Linux

LXD-Nomad 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:

```
$ 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 LXD-Nomad

You should now be able to install LXD-Nomad using:

```
$ pip3 install git+git://github.com/lxd-nomad/lxd-nomad.git
```

Note: Don't have `pip3` installed on your system? Try this:

```
$ sudo apt-get install curl
$ curl https://bootstrap.pypa.io/get-pip.py | sudo python3
```

Command line completion

LXD-Nomad can provide completion for commands 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 LXD-Nomad, you should place the `contrib/completion/bash/nomad` file at `/etc/bash.completion.d/nomad` (or at any other place where your distribution keeps completion files):

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

Make sure to restart your shell before trying to use LXD-Nomad's bash completion.

ZSH

Coming soon!

Your first Nomad file

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

```
name: myproject

containers:
  - name: test01
    image: ubuntu/xenial

  - name: test02
    image: archlinux
```

This Nomad 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:

```
$ nomad 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!

CHAPTER 2

Usage

We don't have proper documentation yet, but if you put a `.nomad.yml` somewhere that looks like:

```
name: myproject
image: debian/jessie
mode: pull # with this mode, debian/jessie is automatically pulled from LXD's public images repo
privileged: true # jessie is systemd

# Those hostnames will be bound to the container's IP in your host's /etc/hosts file
hostnames:
  - myproject.local

# Will mount the project's root folder to /myshare in the container
shares:
  - source: .
    dest: /myshare

# When doing "nomad shell", you'll be having a shell for the specified user/home
shell:
  user: deployuser
  home: /opt/myproject

# Upon our first "nomad up", this ansible playbook will be ran from the host with the container's
# IP in the inventory.
provisioning:
  - type: ansible
    playbook: deploy/site.yml
```

... you should manage to get a workflow similar to Vagrant's.

```
$ nomad up
[ lot's of output because the container is provisioned]
$ curl http://myproject.local
$ nomad shell
# echo "in my container!"
# exit
$ nomad halt
```

It should be noted that the `image` value can also contain a name of a container alias that includes the targetted architecture (eg. `debian/jessie/amd64` or `ubuntu/xenial/armhf`). The image will be pulled from the <https://images.linuxcontainers.org/> image server by default (so you can get a list of supported aliases by using the `lxc image alias list images:` command). You can also choose to use another server by manually setting the `server` value.

Multiple containers

You can define multiple containers in your `.nomad.yml` file.

```
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.

Privileged containers

There seems to be some problems with containers running systemd-based systems. Their init system seem broken. You can confirm this by trying to `exec bash` into the container and try to execute `systemctl`. If you get a dbus-related error, then yup, your container is broken and you need to run the container as privileged.

Shared folders and ACL

Shared folders in LXD-Nomad 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?

For now, the best solution we could come up with is to use ACLs. To ensure that files created by the container are accessible to you back on the host, every new share has a default ACL giving the current user full access to the source folder (`setfacl -Rdm u:<your uid>:rwX <shared source>`).

On the guest side, it's more tricky. LXD-nomad 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. Here's what it could look like:

```
- acl: name=/myshare entity=myuser etype=user permissions=rwX state=present
```

Tired of sudoing for hostname bindings?

Every time a `nomad up` or `nomad halt` is made, we mangle `/etc/hosts` to make our configured hostname bindings work. In a typical setup, your user doesn't have write access to that file. This means that lxd-nomad requires you to type your sudo password all the time. If you're tired of that, give your user write access to `/etc/hosts`.

Sure, there are some security implications in doing that, but on a typical developer box and in this HTTPS Everywhere world, the risk ain't that great.

CHAPTER 3

Command-line reference

Most of your interaction with Nomad will be done using the `nomad` 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. `nomad help` will display help information for the `nomad` command while `nomad help [subcommand]` will show the help for a specifc subcommand. For example:

```
$ nomad help up
usage: LXD-Nomad up [-h] [name [name ...]]

Create, start and provision all the containers of the project according to
your nomad 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
```

nomad config

Command: `nomad config`

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

Options

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

Examples

```
$ nomad config                  # prints project's Nomad file
$ nomad config --containers     # prints project's container names
```

nomad destroy

Command: `nomad 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

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

nomad halt

Command: `nomad 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

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

nomad help

Command: `nomad help [subcommand]`

This command can be used to show help information.

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

Options

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

Examples

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

nomad provision

Command: `nomad 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 LXD-Nomad. That said, the `provision` command can also trigger the execution of provisioning tools that you could've configured in your Nomad file (using the `provisioning` block).

Options

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

Examples

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

nomad shell

Command: `nomad shell [name`

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

Options

- `[name]` - a container name

Examples

```
$ nomad shell mycontainer         # opens a shell into the "mycontainer" container
```

nomad status

Command: `nomad 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

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

nomad up

Command: `nomad 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

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

CHAPTER 4

Nomad file reference

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

Note: LXD-Nomad supports the following names for Nomad files: `.nomad.yml`, `nomad.yml`, `.nomad.yaml` and `nomad.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 Nomad 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 Nomad 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
```

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. LXD-Nomad 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 LXD-Nomad. 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 Nomad 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 LXD-Nomad) 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 LXD-Nomad).

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 LXD-Nomad 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 Nomad 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 LXD-Nomad 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 `nomad up` workflow. This provisioning can also be executed when running `nomad 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
```

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 allows you to define the user to use when doing a `nomad shell`. This allows you to have a shell for a specific user/home directory when doing `nomad shell`:

```
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 LXD-Nomad 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:

```
name: myproject
image: ubuntu/xenial

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

CHAPTER 5

Glossary

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

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 6

Indices and tables

- genindex
- modindex
- search

Index