edi Documentation

Release 1.1.7

Matthias Luescher

Dec 09, 2019

Contents

Contents:

CHAPTER 1

Embedded Development Infrastructure - edi

Driven by the DevOps mindset edi helps you to streamline your embedded development infrastructure. To achieve this goal, edi leverages top-notch open source technologies:

- Ansible is the tool of choice for doing the configuration management.
- LXD allows you to run multiple OS instances on your development host. For complex target system deployments LXD is a great choice too.
- Yaml and Jinja2 are the consistent way to write edi configuration files and Ansible playbooks.
- Python is the language and ecosystem that makes the system integration efficient.
- edi is supposed to be used on the Ubuntu Linux or Debian Linux distribution.
- By default, edi generates Debian Linux based target systems.

1.1 License

edi is licensed under the LGPL license.

1.2 Contributions

You are welcome to contribute to edi. In case of questions you can contact me by e-mail (lueschem@gmail.com).

1.3 More Information

For more information please visit https://www.get-edi.io.

CHAPTER 2

Getting Started

The following setup steps have been tested on Ubuntu 16.04, on Ubuntu 18.04 and on Debian stretch.

2.1 Prerequisites

1. This first step is only required on Ubuntu 16.04 and can be skipped if you are on a more recent Ubuntu or Debian version. edi requires features that got introduced with Ansible 2.1. On Ubuntu 16.04 you can enable xenial-backports and then install Ansible as follows:

   Listing 1: Ubuntu 16.04 only

   ```
   sudo apt install ansible/xenial-backports
   ```

2. Install lxd:

   Listing 2: Ubuntu 16.04

   ```
   sudo apt install lxd/xenial-backports lxcfs/xenial-backports lxd-client/xenial-
   ↪backports liblxc1/xenial-backports
   ```

   Listing 3: Ubuntu 18.04

   ```
   sudo apt install lxd
   ```

   Listing 4: Debian or Ubuntu >= 19.04

   ```
   sudo apt install snapd
   sudo snap install lxd
   sudo usermod -a -G lxd $USER
   ```

3. Close and re-open your user session to apply the new group membership (this guide assumes that you are either member of the group sudoers or admin, for details please read the linux containers documentation).

4. Initialize lxd:

Listing 5: Ubuntu 16.04 or Ubuntu 18.04

```
sudo lxd init
```

Listing 6: Debian or Ubuntu >= 19.04

```
sudo /snap/bin/lxd init
```

The default settings are ok. Use the storage backend "dir" if there is no zfs setup on your computer or if you do not want to create a btrfs pool.

2.2 Installing edi from the Archive

For your convenience, you can directly install edi from a ppa (Ubuntu) or packagecloud (Debian):

1. Add the edi repository to your setup:

Listing 7: Ubuntu

```
sudo add-apt-repository ppa:m-luescher/edi-snapshots
sudo apt-get update
```

Listing 8: Debian

```
curl -s https://packagecloud.io/install/repositories/get-edi/debian/script.deb.sh␣
↪| sudo bash
```

1. Install edi:

```
sudo apt install edi
```

2.3 Setting up ssh Keys

If you plan to access edi generated containers or target systems using ssh, it is a good idea to create a ssh key pair. Hint: edi versions greater or equal than 0.11.0 have a secure by default setup of ssh and disable password based login.

1. Review if you already have existing ssh keys:

```
ls -al ~/.ssh
```

Valid public keys are typically named *id_rsa.pub*, *id_dsa.pub*, *id_ecdsa.pub* or *id_ed25519.pub*.

2. If there is no valid ssh key pair, generate one:

```
$ ssh-keygen -t rsa -b 4096 -C "you@example.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/YOU/.ssh/id_rsa):
Created directory '/home/YOU/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

Hint: If you decided to use a passphrase and do not want to reenter it every time, it is a good idea to use a *ssh-agent*.

2.4 Building a First Container

1. Create an empty project folder:

```
cd ~/
mkdir my-first-edi-project
cd my-first-edi-project
```

2. Generate a configuration for your project:

```
edi config init my-project debian-stretch-amd64
```

3. Build your first (development) lxc container named *my-first-edi-container*:

```
sudo edi -v lxc configure my-first-edi-container my-project-develop.yml
```

2.5 Exploring the Container

1. Log into the container using your current user name (Note: This user is only available within a development container.) and the password *ChangeMe!*:

```
lxc exec my-first-edi-container -- login ${USER}
```

2. Change the password for your container user:

```
passwd
```

3. Install a package within the container:

```
sudo apt install cowsay
```

4. Share a file with the host (Note: The folder ~/edi-workspace is shared with your host.):

```
cowsay "Hello world!" > ~/edi-workspace/hello
```

5. Leave the container:

```
exit
```

6. Read the file previously created within the container:

```
cat ~/edi-workspace/hello
```

7. Enter the container as root (Note: This is useful if you have a container without your personal user.):

```
lxc exec my-first-edi-container -- bash
```

8. And leave it again:

```
exit
```

9. Get the IP address of the container:

```
lxc list my-first-edi-container
```

10. Enter the container using ssh:

```
ssh CONTAINER_IP
```

11. And leave it again:

```
exit
```

CHAPTER 3

Working with the edi Source Code

Instead of installing edi from the ppa you can also work directly with the source code.

1. Clone the source code:

```
git clone https://github.com/lueschem/edi.git
```

2. Change into the edi subfolder:

```
cd edi
```

3. Install various packages that are required for the development of this project:

```
sudo apt install -y git-buildpackage dh-make equivs && sudo mk-build-deps -i␣
↪debian/control
```

4. Build the edi Debian package (just to verify that everything works):

```
debuild -us -uc
```

5. Make the development setup convenient by adding some environment variables (they are only valid for the current shell):

```
source local_setup
```

6. Verify that the source code version of edi is being used:

```
which edi
edi version
```

CHAPTER 4

Configuration Management

4.1 Introduction

The management of complex embedded software product line projects is a main focal point of edi. Such projects may be managed by many people that are spread over the world. Maintaining a reproducible environment for all involved parties is a key success factor for such projects.

edi will help the different stakeholders to manage their use cases. Here is an example with four stakeholders:

- The *developer* needs a system that comes with development tools, libraries, header files etc. Also an integrated development environment (IDE) might be part of his wish list. A pre configured user account is an additional plus.
- The *maintainer of the CI server* needs a similar setup like the developer. However - to speed up the build process - he might want to use images that come without a heavy weight IDE.
- In theory, the *tester* should do his tests on a production image. Unfortunately production images might be hardened and therefore the tester is unable to do some introspection of the system. Therefore the tester is actually asking for a production image with some "add-ons" like ssh access and a simple editor.
- The *operator* wants a rock solid production image with all development back doors removed. Logging output should be reduced to a minimum to protect the flash storage.

All involved parties have the common concern that they want to maintain consistency across the whole project(s). edi achieves this by managing the different use cases with a single project setup. The following four pillars are in place to enable reusability and extensibility, reduce duplicate code and guarantee consistency:

- *Yaml Based Configuration*: The whole project configuration is written in yaml. Yaml is easy to read and write for both humans and machines.
- *Jinja2*: Sometimes there is a need to parametrize parts of the configuration. The jinja2 template engine allows you to do this.
- *Overlays*: Depending on your use case you might want to change some specific aspects of the project configuration. The overlays allow you to customize a use case globally, per user or per host machine.

- *Plugins*: While every embedded project is somehow different, they all share some commonalities. Plugins make the commonalities shareable among multiple projects while they allow the full customization of the unique features of a project.

4.2 Yaml Based Configuration

Within an empty directory the following command can be used to generate an initial edi configuration:

```
edi config init my-project debian-stretch-amd64
```

This command generates a configuration with four placeholder use cases:

- *my-project-run.yml*: This configuration file covers the *run* use case. It is the configuration that the customer will get.
- *my-project-test.yml*: The *test* use case shall be as close as possible to the *run* use case. A few modifications that enable efficient testing will differentiate this use case from the *run* use case.
- *my-project-build.yml*: The *build* use case covers the requirements of a build server deployment.
- *my-project-develop.yml*: The *develop* use case satisfies the requirements of the developers.

Please note that the above use cases are just an initial guess. edi does not at all force you to build your project upon a predefined set of use cases. It just helps you to modularize your different use cases so that they do not diverge over time.

The configuration is split into several sections. The following command will dump the merged and rendered configuration of the use case *develop* for the given command:

```
edi lxc configure --config my-container my-project-develop.yml
```

4.2.1 `general` Section

The general section contains the information that might affect all other sections.

edi supports the following settings:

key	description
edi_compression	The compression that will be used for edi (intermediate) artifacts. Possible values are `gz` (fast but not very small), `bz2` or `xz` (slower but minimal required space). If not specified, edi uses `xz` compression.
edi_lxc_stop_timeout	The maximum time in seconds that edi will wait until it forces the shutdown of the lxc container. The default timeout is `120` seconds.
edi_required_minimal_edi_version	Defines the minimal edi version that is required for the given configuration. If the edi executable does not meet the required minimal version, it will exit with an error. If not specified, edi will not enforce a certain minimal version. A valid version string value looks like `0.5.2`.
edi_lxc_network_interface_name	The default network interface that will be used for the lxc container. If unspecified edi will name the container interface `lxcif0`.
edi_config_management_user	The target system user that will be used for configuration management tasks. Please note that direct lxc container management uses the root user. If unspecified edi will name the configuration management user `edicfgmgmt`.
parameters	Optional general parameters that are globally visible for all plugins. Parameters need to be specified as key value pairs.

4.2.2 `bootstrap` Section

This section tells edi how the initial system shall be bootstrapped. The following settings are supported:

key	description
architecture	The architecture of the target system. For Debian possible values are any supported architecture such as `amd64`, `armel` or `armhf`.
repository	The repository specification where the initial image will get bootstrapped from. A valid value looks like this: `deb http://deb.debian.org/debian/ buster main`.
repository_key	The signature key for the repository. *Attention*: If you do not specify a key the downloaded packages will not be verified during the bootstrap process. *Hint*: It is a good practice to download such a key from a https server. A valid repository key value is: `https://ftp-master.debian.org/keys/archive-key-9.asc`.
tool	The tool that will be used for the bootstrap process. Currently only `debootstrap` is supported. If unspecified, edi will choose `debootstrap`.
additional_packages	A list of additional packages that will be installed during bootstrapping. If unspecified, edi will use the following default list: `['python', 'sudo', 'netbase', 'net-tools', 'iputils-ping', 'ifupdown', 'isc-dhcp-client', 'resolvconf', 'systemd', 'systemd-sysv', 'gnupg']`.

Please note that edi will automatically do cross bootstrapping if required. This means that you can for instance bootstrap an armhf system on an amd64 host.

If you would like to bootstrap an image right now, you can run the following command:

```
sudo edi image bootstrap my-project-develop.yml
```

4.2.3 `qemu` Section

If the target architecture does not match the host architecture edi uses QEMU to emulate the foreign architecture. edi automatically detects the necessity of an architecture emulation and takes the necessary steps to set up QEMU. As QEMU evolves quickly it is often desirable to point edi to a very recent version of QEMU. The QEMU section allows you to do this. The following settings are available:

key	description
package	The name of the qemu package that should get downloaded. If not specified edi assumes that the package is named `qemu-user-static`.
repository	The repository specification where QEMU will get downloaded from. A valid value looks like this: `deb http://deb.debian.org/debian/ stretch main`. If unspecified, edi will try to download QEMU from the repository indicated in the bootstrap section.
repository_key	The signature key for the QEMU repository. *Attention*: If you do not specify a key the downloaded QEMU package will not be verified. *Hint*: It is a good practice to download such a key from a https server. A valid repository key value is: `https://ftp-master.debian.org/keys/archive-key-8.asc`.

4.2.4 Ordered Node Section

In order to understand the following sections we have to introduce the concept of an *ordered node section*. In Unix based systems it is quite common to split configurations into a set of small configuration files (see e.g. `/etc/sysctl.d`). Those small configuration files are loaded and applied according to their alphanumerical order. edi does a very similar thing in its *ordered node sections*. Here is an example:

Listing 1: Example 1

```
dog_tasks:
  10_first_task:
    job: bark
  20_second_task:
    job: sleep
```

Listing 2: Example 2

```
dog_tasks:
  20_second_task:
    job: sleep
  10_first_task:
    job: bark
```

In both examples above the dog will first bark and then sleep because of the alphanumerical order of the nodes `10_first_task` and `20_second_task`. The explicit order of the nodes makes it easy to add or modify a certain node using *Overlays*.

4.2.5 Plugin Node

Most of the ordered node sections contain nodes that specify and parametrize plugins.

A typical node looks like this:

```
lxc_profiles:
  10_first_profile:
     path: path/to/profile.yml
     parameters:
       custom_param_1: foo
       custom_param_2: bar
```

Such nodes accept the following settings:

key	description
path	A relative or absolute path. Relative paths are first searched within `edi_project_plugin_directory` and if nothing is found the search falls back to `edi_edi_plugin_directory`. The values of the plugin and project directory can be retrieved as follows: `edi lxc configure --dictionary SOME-CONTAINER SOME_CONFIG.yml`.
parameters	An optional list of parameters that will be used to parametrize the given plugin.
skip	`True` or `False`. If `True` the plugin will not get applied. If unspecified, the plugin will get applied.

To learn more about plugins please read the chapter *Plugins*.

4.2.6 `lxc_templates` Section

The lxc_templates section is an *ordered node section* consisting of *plugin nodes*. Please consult the LXD documentation if you want to write custom templates.

4.2.7 `lxc_profiles` Section

The lxc_profiles section is an *ordered node section* consisting of *plugin nodes*. Please consult the LXD documentation if you want to write custom profiles.

4.2.8 `playbooks` Section

The playbooks section is an *ordered node section* consisting of *plugin nodes*. Please consult the Ansible documentation if you want to write custom playbooks.

4.2.9 `postprocessing_commands` Section

The postprocessing_commands section is an *ordered node section* consisting of *plugin nodes*. The post processing commands can be written in any language of choice. In contrast to the other plugin nodes the post processing command nodes require an explicit declaration of the generated artifacts. Please read the chapter *Plugins* for more details.

4.2.10 `shared_folders` Section

The shared_folders section is an *ordered node section* that can be used to specify shared folders between LXC containers and their host.

Shared folders are very convenient for development use cases. Please note that edi will automatically turn any container that uses shared folders into a *privileged* container. This will facilitate the data exchange between the host and the target system. It is advisable to use shared folders together with the development_user_facilities playbook plugin.

A shared folder section can look like this:

```
shared_folders:
  edi_workspace:
    folder: edi-workspace
    mountpoint: edi-workspace
```

Let us assume that the name of the current development user is `johndoe` and that his home directory is `/home/johndoe`. The development_user_facilities playbook plugin will automatically make sure that the user `johndoe` will also exist within the container. The shared_folders section will then make sure that the host folder `/home/johndoe/edi-workspace` (`folder`) will be shared with the container using the container directory `/home/johndoe/edi-workspace` (`mountpoint`).

The shared folder nodes accept the the following settings:

key	description
folder	The name of the host folder within the home directory of the current user. If the folder does not exist, edi will create it.
mountpoint	The name of the mount point within the container home directory of the current user. If the mount point does not exist edi will display an error. *Hint*: It is assumed that the mount points within the container will get created using an appropriate playbook. The development_user_facilities playbook plugin will for instance take care of mount point creation.
skip	`True` or `False`. If `True` the folder will not be shared. If unspecified, the folder will get shared.

4.3 Jinja2

A closer look at the configuration created in the previous chapter reveals some parametrization: The file `my-project-develop.yml` contains a line that dynamically derives the name of an artifact from the project name (`sample_output: {{ edi_configuration_name }}.result`). Jinja2 will replace the expression `{{ edi_configuration_name }}` with the name of the configuration.

The following command can be used to display the dictionary that is available for Jinja2 operations when loading the configuration `my-project-develop.yml`:

```
edi image create --dictionary my-project-develop.yml
```

Since the dictionary is context sensitive to the sub-command you have to specify the full command with the additional option `--dictionary` to display the appropriate dictionary. The option `--dictionary` is available for all commands that deal with configuration.

`my-project-develop.yml` contains an even more complicated parametrization in the `lxc_profiles` section:

```
{% if edi_lxd_version is defined and (edi_lxd_version.split('.')[0] | int >= 3 or edi_
↪lxd_version.split('.')[1] | int >= 9) %}
  200_default_root_device:
    path: lxc_profiles/general/default_root_device/default_root_device.yml
{% endif %}
```

This conditional code will make sure that an additional LXD profile gets generated and applied for recent LXD versions.

Plugins can even further benefit from Jinja2 since there are additional dictionary entries available. The option `--plugins` will output the details:

```
edi image create --plugins my-project-develop.yml
```

If supported for the plugin, `edi` will preview the plugin rendered by Jinja2 when using the above command. Given the plugin is an Ansible playbook, the whole plugin dictionary will be made available to the playbook by means of the Ansible command line option `--extra-vars`.

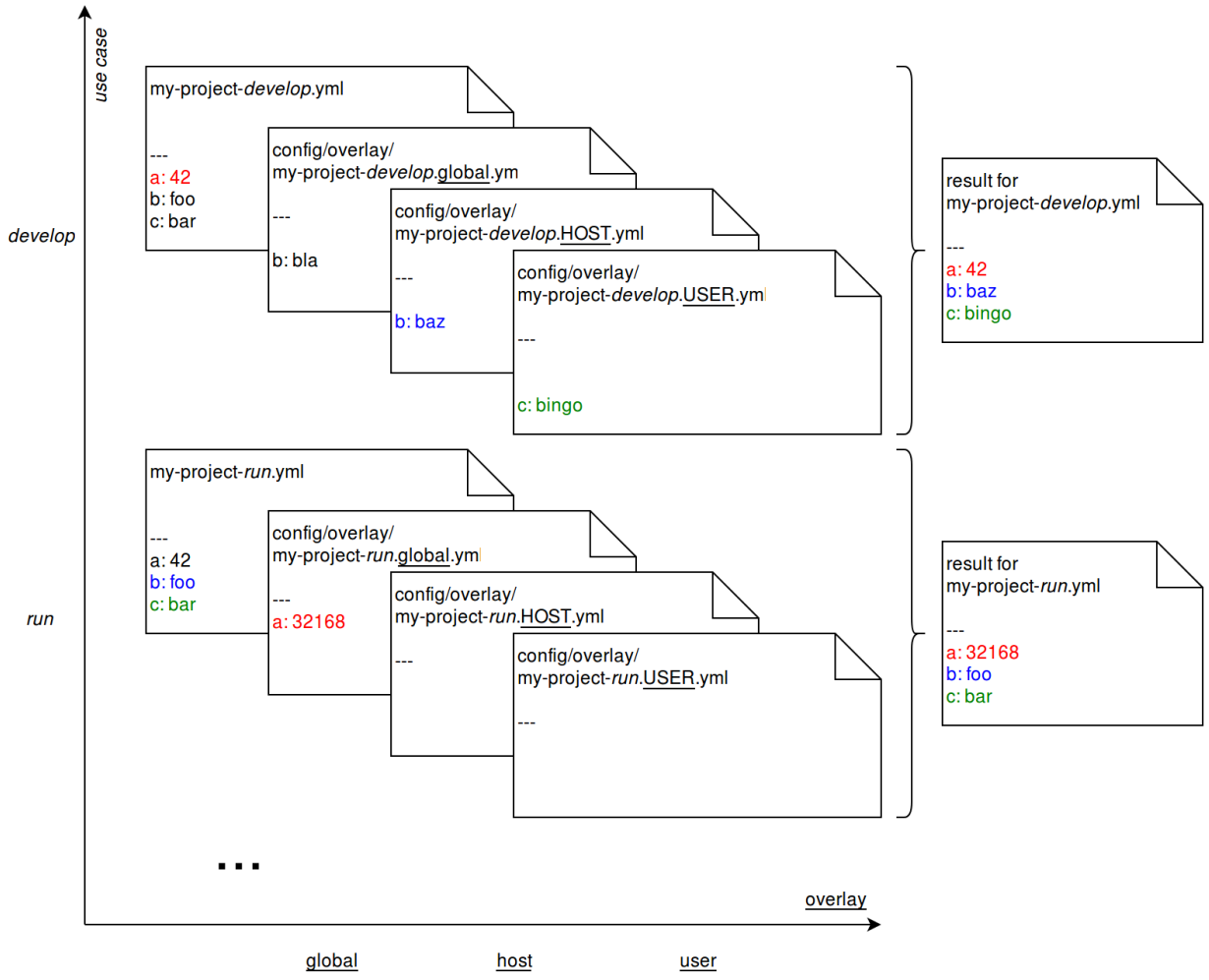
4.4 Overlays

As soon as a single `edi` project configuration should support multiple use cases the use of overlays will help to get rid of duplicate configuration code. When using overlays, it is a good practice to put most of the configuration code into a single yaml file. In the example configuration used throughout the previous chapters this is the file `configuration/base/my-project.yml`. A use case like `my-project-develop.yml` is then just a symbolic link to this configuration file. The differentiation between the use cases happens in the `global` overlay (e.g. `configuration/overlay/my-project-develop.global.yml`): `edi` will initially load the base configuration and then merge it with the `global` overlay. The configuration done in the `global` overlay takes precedence over the configuration done in the base configuration.

`edi` furthermore supports two additional overlays: The configuration can be further tuned per `host` (the overlay file shall then end with `.$(hostname).yml`, e.g. `.buildd.yml`) and per `user` (the overlay file shall then end with `.$(id -un).yml`, e.g. `.johndoe.yml`). The `user` overlay takes the highest precedence.

The following picture illustrates how yaml configuration files will get merged:

The merged configuration can be displayed using a command like:


use case
develop
run
my-project-develop.yml
---
a: 42
b: foo
c: bar
config/overlay/
my-project-develop.global.ym
---
b: bla
config/overlay/
my-project-develop.HOST.yml
---
b: baz
config/overlay/
my-project-develop.USER.ym
---
c: bingo
result for
my-project-develop.yml
---
a: 42
b: baz
c: bingo
my-project-run.yml
---
a: 42
b: foo
c: bar
config/overlay/
my-project-run.global.ym
---
a: 32168
config/overlay/
my-project-run.HOST.yml
---
config/overlay/
my-project-run.USER.yml
---
result for
my-project-run.yml
---
a: 32168
b: foo
c: bar
overlay
global
host
user

```
edi lxc configure --config my-dev-container my-project-develop.yml
```

The usage of overlays is optional and in any case it is not necessary to specify all possible overlays.

4.5 Plugins

edi comes with a few reusable plugins:

4.5.1 LXC/LXD Templates

During the root file system assembly edi adds templates to the container image (see LXD Documentation).

The following templates are already predefined:

Hostname

This template dynamically adds the `/etc/hostname` file to the container.

Listing 3: Configuration Example

```
lxc_templates:
  ...
  100_etc_hostname:
    path: lxc_templates/debian/hostname/hostname.yml
  ...
```

Hosts

This template dynamically adds the `/etc/hosts` file to the container.

Listing 4: Configuration Example

```
lxc_templates:
  ...
  200_etc_hosts:
    path: lxc_templates/debian/hosts/hosts.yml
  ...
```

4.5.2 LXC/LXD Profiles

With the help of profiles a container configuration can be fine tuned in a modular way (see LXD Profile Documentation).

The following profiles have proven to be useful for various projects:

Default Network Interface

This profile adds a default network interface to the container named according to the value of `edi_lxc_network_interface_name`. The interface is of type `bridged` and its parent is `lxdbr0`.

Listing 5: Configuration Example

```
lxc_profiles:
  ...
  100_lxc_networking:
      path: lxc_profiles/general/lxc_networking/default_interface.yml
  ...
```

Default Root Device

This profile makes sure that the container uses the `default` storage pool as its root device. Please note that newer LXD versions (>=2.9) require the configuration of a storage pool.

Listing 6: Configuration Example

```
lxc_profiles:
  ...
  200_default_root_device:
    path: lxc_profiles/general/default_root_device/default_root_device.yml
  ...
```

Privileged Mode

This profile will make sure that the container is running in privileged mode.

Listing 7: Configuration Example

```
lxc_profiles:
  ...
  300_privileged:
    path: lxc_profiles/general/security/privileged.yml
  ...
```

Please note that if a container has one or more *shared folders* configured it will automatically be turned into privileged mode.

Suppress Init

This profile will make sure that the container does not start using systemd but instead uses dumb-init. This is especially useful during the build of a distributable image. During such a build you just want to assemble the image without starting any services.

The following configuration snippet will conditionally enable the usage of dumb-init:

Listing 8: Configuration Example

```
lxc_profiles:
  ...
  400_suppress_init:
    path: lxc_profiles/general/suppress_init/suppress_init.yml
    skip: {{ not edi_create_distributable_image }}
  ...
```

dumb-init is not part of the default package set during bootstrapping. For this reason you have to add it within the bootstrap section (otherwise the launching of the container will fail):

Listing 9: Configuration Example

```
bootstrap:
  ...
  additional_packages: ["python", "sudo", "netbase", "net-tools", "iputils-ping",
↪"ifupdown", "isc-dhcp-client", "resolvconf", "systemd", "systemd-sysv", "gnupg",
↪"dumb-init"]
  ...
```

4.5.3 Ansible Playbooks

edi ships with a few Ansible playbooks that can be re-used in many projects. This playbooks can also serve as an example if you want to write a custom playbook for your own project.

Please take a look at the comprehensive documentation of Ansible if you want to write your own playbook.

Here is a description of the built-in playbooks including the parameters that can be used to fine tune them:

Base System

The base system playbook tackles the following tasks:

- Setup the lxc container network interface (optional).
- Inherit the proxy settings from the host computer (optional).
- Perform a basic apt setup.
- Add a default user (optional).
- Install an openssh server (optional).

The following code snippet adds the base system playbook to your configuration:

Listing 10: Configuration Example

```
playbooks:
  ...
  100_base_system:
    parameters:
      create_default_user: true
      install_openssh_server: true
    path: playbooks/debian/base_system/main.yml
  ...
```

The playbook can be fine tuned as follows:

key	description
apply_proxy_settings	With this boolean value you can specify if the target system shall get a proxy setup. The default value is `True` and the standard behavior is that the target system will inherit the proxy settings of the host system. However, the proxy settings can be customized according to the table below. If you specify `False` the target system proxy setup will remain untouched.
configure_lxc_network_interface	By default (boolean value `True`) the playbook will add a lxc network interface to the container. If this behavior is not desired, change the setting to `False`.
create_default_user	By default (boolean value `False`) no additional user gets created. If you need an additional user switch this value to `True` and fine tune the default user according to the table below.
install_openssh_server	By default (boolean value `False`), no ssh server will be installed on the target system. Switch this value to `True` if you would like to access the system using ssh.
disable_ssh_password_authentication	By default password authentication is disabled for ssh (boolean value `True`). If you want to allow password based authentication then switch this value to `False` but make sure to use a non standard password.
authorize_current_user	By default (boolean value `True`) the current host user will be authorized to ssh into the account of the default user. Switch this value to `False` if the current user shall not be authorized.
ssh_pub_key_directory	All the public keys (ending with .pub) contained in the folder `ssh_pub_key_directory` (defaults to `{{ edi_project_directory }}/ssh_pub_keys`) will be added to the list of authorized ssh keys of the default user.
install_documentation	By default (boolean value `True`) the documentation of every Debian package will get installed. Switch this value to `False` if you want to deploy an image with a minimal footprint.
translations_filter	By default all translations contained in Debian packages will get installed (empty filter: `""`). To reduce the footprint of the resulting artifacts the number of installed languages can be limited. By choosing the builtin filter `"en_translations_only"` you can make sure that only English translations will get installed.

The proxy settings can be customized as follows:

key	description
target_http_proxy	The http proxy that gets applied to the target system (defaults to `{{ edi_host_http_proxy }}`).
target_https_proxy	The https proxy that gets applied to the target system (defaults to `{{ edi_host_https_proxy }}`).
target_ftp_proxy	The ftp proxy that gets applied to the target system (defaults to `{{ edi_host_ftp_proxy }}`).
target_socks_proxy	The socks proxy that gets applied to the target system (defaults to `{{ edi_host_socks_proxy }}`).
target_no_proxy	The proxy exception list that gets applied to the target system (defaults to `{{ edi_host_no_proxy }}`).

The default user can be fine tuned as follows:

key	description
default_user_group_name	The group name of the default user (default is `edi`).
default_user_gid	The group id of the default user (default is `2000`).
default_user_name	The user name of the default user (default is `edi`).
default_user_uid	The user id of the default user (default is `2000`).
default_user_shell	The shell of the default user (default is `/bin/bash`).
default_user_groups	The groups of the default user (default is `adm,sudo`).
default_user_password	The initially set password of the default user (default is `ChangeMe!`).

Base System Cleanup

The base system cleanup playbook makes sure that we get a clean distributable image by doing the following tasks:

- It removes the openssh server keys (they shall be unique per system).
- It removes cached apt data to reduce the artifact footprint.
- It finalizes the proxy setup.
- It sets the final hostname.

The following code snippet adds the base system cleanup playbook to your configuration:

Listing 11: Configuration Example

```
playbooks:
  ...
  900_base_system_cleanup:
      path: playbooks/debian/base_system_cleanup/main.yml
      parameters:
          hostname: raspberry
  ...
```

The playbook can be fine tuned as follows:

key	description
hostname	Set the hostname within the final artifact (default is `edi`).
regenerate_openssh_server_keys	By default the playbook will make sure that the openssh server keys get regenerated (boolean value `True`). Switch this value to `False` if you would like to keep the same openssh server keys for all instances that will receive this artifact.
cleanup_proxy_settings	By default the proxy settings of the resulting artifact will get cleaned up (boolean value `True`). If you would like to keep the same proxy settings switch this value to `False`. When set to `True`, the proxy settings can be fine tuned according to the table below.

The final proxy settings can be customized as follows:

key	description
target_http_proxy	The final http proxy settings (defaults to `""`).
target_https_proxy	The final https proxy settings (defaults to `""`).
target_ftp_proxy	The final ftp proxy settings (defaults to `""`).
target_socks_proxy	The final socks proxy settings (defaults to `""`).
target_no_proxy	The final proxy exception list (defaults to `""`).

Development User Facilities

The development user facilities playbook adds the host user (the user that runs `edi`) to the target system. In case the target system is an LXD container and shared folders are defined, the playbook will make sure that the specified folders are shared between the host system and the LXD container.

The host user will automatically be authorized to ssh into the target system.

The password for the user (same user name as the host user) in the target system will be `ChangeMe!`.

Please note that this playbook will get skipped entirely when a distributable image gets created (when `edi_create_distributable_image` is `True`).

The following code snippet adds the development user facilities playbook to your configuration:

Listing 12: Configuration Example

```
playbooks:
  ...
  200_development_user_facilities:
      path: playbooks/debian/development_user_facilities/main.yml
  ...
```

4.5.4 Postprocessing Commands

Postprocessing commands can be used to gradually transform an exported LXD container into the desired artifacts (e.g. an image that can get flashed to an SD card).

A typical post processing command can be configured as follows:

Listing 13: Configuration Example

```
postprocessing_commands:
  ...
  100_lxd2rootfs:
      path: postprocessing_commands/rootfs/lxd2rootfs.edi
      require_root: True
      output:
          pi3_rootfs: {{ edi_configuration_name }}_rootfs
  ...
```

`edi` will render the file `postprocessing_commands/rootfs/lxd2rootfs.edi` using the Jinja2 template engine and then execute it. It is a good practice to use this file as a thin shim between `edi` and the scripts that do the heavy lifting.

The statement `require_root: True` tells edi that a privileged user (sudo) is needed to execute the command.

Each post processing command shall create at least one (intermediate) artifact that gets specified within the `output` node. The resulting artifact can be used as an input for the next post processing command.

The specified output can be either a single file or a folder (if multiple files get generated by the command).

The variable `edi_input_artifact` can be used to locate the artifact that got generated before the post processing commands get called. It contains typically the artifact created by the `edi lxc export` command.

The post processing commands are implemented in a very generic way and to get an idea of what they can do please take a look at the the edi-pi configuration.

CHAPTER 5

Command Pipeline

edi is designed to divide big tasks into small sub commands. Each sub command will initiate the transition into a new state of available artifacts:

If the desired original state has not yet been reached, edi will make sure that all necessary sub commands get executed to reach the desired state.

Example:

The following command will make sure that - after a successful execution - a fully configured lxd container is available:

```
sudo edi lxc configure NAME CONFIG
```

If the intermediate artifacts are to some degree not available, edi will execute all required sub commands - if needed it will start with the *image bootstrap* sub command.

Please note that the intermediate artifacts are not checked if they are fully up to date. If you want to make sure that all intermediate artifacts for a given configuration get recreated then execute the following command:

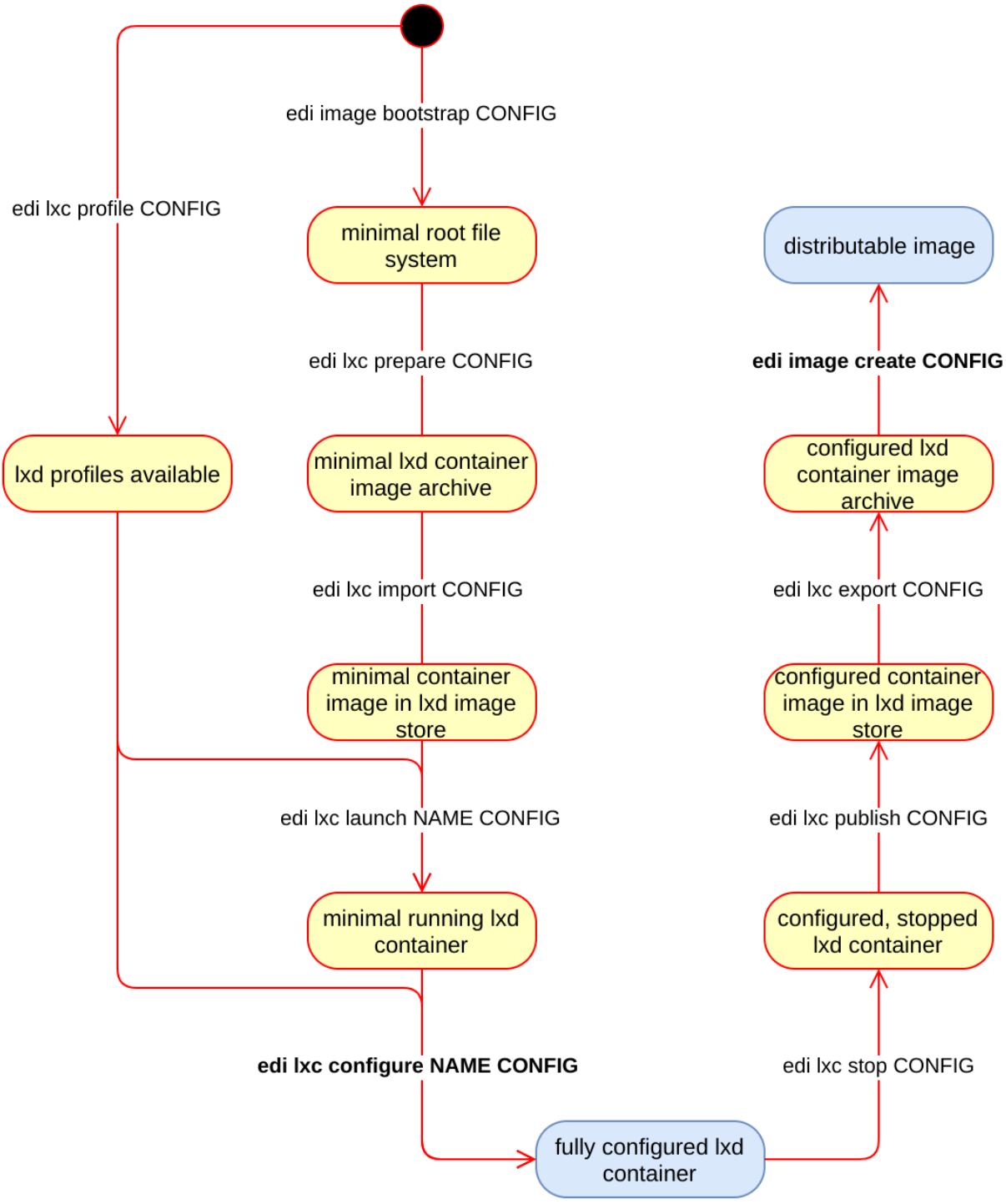
```
edi clean CONFIG
```

The above command will delete the previously generated artifacts. However, it will not delete named lxd containers.

Depending upon the selected command, `edi` will set the following two useful parameters:

`edi_configure_remote_target` will be set to `True` if a remote system gets configured.

`edi_create_distributable_image` will be set to `True` it a distributable image gets created. This is useful to skip certain steps shall not be applied to a distributable image or to include additional steps that are only needed in case of a distributable image.


edi image bootstrap CONFIG
edi lxc profile CONFIG
minimal root file system
distributable image
edi lxc prepare CONFIG
edi image create CONFIG
lxd profiles available
minimal lxd container image archive
configured lxd container image archive
edi lxc import CONFIG
edi lxc export CONFIG
minimal container image in lxd image store
configured container image in lxd image store
edi lxc launch NAME CONFIG
edi lxc publish CONFIG
minimal running lxd container
configured, stopped lxd container
edi lxc configure NAME CONFIG
edi lxc stop CONFIG
fully configured lxd container

CHAPTER 6

Upgrade Notes

6.1 LXD Storage Pool

Newer lxd versions (>=2.9) require the configuration of a storage pool. edi (>=0.6.0) ships with a plugin for a default storage pool. You can add the following lines to the lxc_profiles section of your existing configuration if you want to upgrade to a newer version of lxd:

```
lxc_profiles:

  ...

{% if edi_lxd_version is defined and (edi_lxd_version.split('.')[0] | int >= 3 or edi_
↪lxd_version.split('.')[1] | int >= 9) %}
  020_default_root_device:
    path: lxc_profiles/general/default_root_device/default_root_device.yml
{% endif %}

  ...
```

Please note that newly created configurations will already contain this conditional inclusion of the storage pool definition. If the above configuration is missing, `edi lxc configure ...` will print an error message:

```
$ sudo edi -v lxc configure my-project my-project-test.yml
...
Going to launch container.
INFO:root:Running command: ['sudo', '-u', 'lueschm1', 'lxc', 'launch', 'local:my-
↪project-test_edicommand_lxc_import', 'my-project', '-p', 'lxcif0_
↪0c4a88500d0670949c8f']
Creating my-project
Error: Launching image 'my-project-test_edicommand_lxc_import' failed with the␣
↪following message:
error: No root device could be found.
```

On Ubuntu 16.04 the following command can be used to upgrade the lxd installation:

```
sudo apt install lxd/xenial-backports lxd-client/xenial-backports
```

CHAPTER 7

Performance Tuning

7.1 Enable Ansible Pipelining

Ansible can be switched into pipelining mode when executing playbooks. This can significantly increase the performance especially when using emulated environments.

To enable pipelining just add `ansible_pipelining: true` to the `general/parameters` section of your project configuration:

Listing 1: Ansible pipelining

```
general:
  ...
  parameters:
    ...
    ansible_pipelining: true
    ...
...
```

7.2 Choosing a Suitable Compression Algorithm

A lot of intermediate artifacts of edi get compressed. The default compression algorithm is `xz`. The `xz` algorithm is very good at reaching a high compression rate but it is rather slow. To get some more speed when doing frequent builds it is advisable to switch to the `gz` algorithm.

This can be done within the `general` section of the project configuration:

Listing 2: Compression algorithm

```
general:
  ...
```

(continues on next page)

(continued from previous page)

```
  edi_compression: gz
...
```

7.3 Avoid Re-bootstrapping

The bootstrapping process using `debootstrap` is pretty time consuming - especially when doing it for a foreign architecture. In most cases the bootstrapped artifact is not affected by modifications done to the project configuration. Therefore it is in most of the cases OK to keep the bootstrapped artifact when doing a next build. The tool `edi` supports this workflow through the `--recursive-clean NUMBER` command line option. Please take a look at this blog post for a detailed example.

7.4 Re-configure your Container Instead of Re-creating it

The tool `edi` enables you to do a lot of development work within a container that is very similar to the target device. As the project configuration will change over time, also the development container should be changed accordingly. Luckily the container setup can be adjusted by just re-executing the command that got used in first place to generate the container (e.g. `edi -v lxc configure CONTAINERNAME CONFIG.yml`).

CHAPTER 8

Reference List

8.1 edi Blog

The following blog posts complement this documentation:

- Compiling For Embedded Debian Target Systems
- Cross Compiling For Raspbian
- A New Approach To Operating System Image Generation
- Secure By Default Ssh Setup
- 11 Traps To Avoid When Building Debian Images
- Running Gui Applications Within Lxd Container
- Booting Debian with U-Boot
- Updating a Debian Based IoT Fleet with Mender

8.2 Debian

- Debian Python Policy
- Building Debian Packages with git

8.3 Python

8.3.1 Packaging

- Packaging and Distributing Projects using setuptools.
- Using setuptools_scm to derive version from git tag.

8.3.2 Documentation

- Sphinx - a Python documentation generator.

8.3.3 Libraries

- "Requests is an elegant and simple HTTP library for Python, built for human beings."
- Jinja2 is a template engine for Python.

8.4 LXC/LXD

- The LXD blog gives a very good introduction to lxc/lxd 2.0.

8.5 Restructured Text

- Wikipedia about ReStructuredText.
- reStructuredText Markup Specification.

CHAPTER 9

Command Cheat Sheet

9.1 edi

Enable bash completion during development and add the edi bin folder to the PATH:

```
source local_setup
```

Run the short tests (including coverage):

```
py.test-3 --cov=edi --cov-report=html
```

Run all tests (including coverage):

```
sudo py.test-3 --all --cov=edi --cov-report=html
```

Check source code using flake8:

```
flake8 --max-line-length=120 .
```

9.2 Debian

Build an edi .deb package directly:

```
debuild -us -uc
```

Build an edi .deb package using git-buildpackage:

```
gbp buildpackage
```

Install the resulting package:

```
sudo dpkg -i ../edi_X.X.X_all.deb
```

9.3 Python

Create a source distribution of edi:

```
python3 setup.py sdist
```

Install edi in editable mode (development setup):

```
pip3 install -e .
```

9.4 Documentation

Build the shinx html documentation of edi:

```
cd docs && make html
```

9.5 git

Initial personalization of git:

```
git config --global user.email "johndoe@example.com"
git config --global user.name "John Doe"
```

CHAPTER 10

Presentations

10.1 Efficient Usage of Debian on Embedded Devices

Location: OSADL Networking day

Date: 5. June 2019

Slides: EfficientDebian.pdf

10.2 Real Time Linux

Location: Embedded GNU/Linux Developer Meetup

Date: 3. September 2018

Slides: RealTimeLinux.pdf

10.3 Cross Compiling For Embedded Debian Target Systems

Location: Embedded GNU/Linux Developer Meetup

Date: 2. July 2018

Slides: DebianCross.pdf