Rockcraft

Release 1.3.1

Canonical Ltd.

Apr 23, 2024

CONTENTS

Rockcraft is a tool to create *rocks* – a new generation of secure, stable and OCI-compliant container images, based on Ubuntu.

Rockcraft is for anyone who wants to build production-grade container images, regardless of their experience as a software developer – from independent software vendors to cloud-native developers and occasional container users. Rockcraft handles all the repetitive and boilerplate steps of a build, directing your focus to what really matters: the image's content.

Using the same language as Snapcraft and Charmcraft, Rockcraft offers a truly declarative way for building efficient container images. By making use of existing Ubuntu tools like LXD and Multipass, Rockcraft is able to compartmentalise typical container image builds into multiple parts, each one being comprised of several independent lifecycle steps, allowing complex operations to be declared at build time.

CHAPTER

ONE

TUTORIALS

If you want to learn the basics from experience, then our tutorials will help you acquire the necessary competencies from real-life examples with fully reproducible steps.

The first tutorials will familiarise you with the basic operations, tools and workflows for packing rocks.

1.1 Create a “Hello World” rock

1.1.1 Prerequisites

- snap enabled system (https://snapcraft.io)
- LXD installed (https://linuxcontainers.org/lxd/getting-started-cli/)
- skopeo installed (https://github.com/containers/skopeo)
- Docker installed (https://snapcraft.io/docker)
- a text editor

1.1.2 Install Rockcraft

Install Rockcraft on your host:

```
sudo snap install rockcraft --classic
```

1.1.3 Project Setup

Create a new directory and write the following into a text editor and save it as `rockcraft.yaml`:

```
# Metadata section

name: hello
summary: Hello World
description: The most basic example of a rock.
version: "latest"
license: Apache-2.0

base: bare
build-base: ubuntu@22.04
```

(continues on next page)

(continued from previous page)

```
platforms:
  amd64:  # Make sure this value matches your computer's architecture

# Parts section

parts:
  hello:
    plugin: nil
    stage-packages:
      - hello
```

This file instructs Rockcraft to build a rock that **only** has the `hello` package (and its dependencies) inside. For more information about the `parts` section, check *Part properties*. The remaining YAML fields correspond to metadata that help define and describe the rock. For more information about all available fields check *rockcraft.yaml*.

1.1.4 Pack the rock with Rockcraft

To build the rock, run:

```
rockcraft pack
```

The output should look as follows:

```
Launching instance...
Retrieved base bare for amd64
Extracted bare:latest
Executed: pull hello
Executed: pull pebble
Executed: overlay hello
Executed: overlay pebble
Executed: build hello
Executed: build pebble
Executed: stage hello
Executed: stage pebble
Executed: prime hello
Executed: prime pebble
Executed parts lifecycle
Exported to OCI archive 'hello_latest_amd64.rock'
```

At the end of the process, a file named `hello_latest_amd64.rock` should be present in the current directory. That's your rock, in oci-archive format (a tarball).

1.1.5 Run the rock in Docker

First, import the recently created rock into Docker:

```
sudo /snap/rockcraft/current/bin/skopeo --insecure-policy copy oci-archive:hello_latest_
↪amd64.rock docker-daemon:hello:latest
```

Now run the `hello` command from the rock:

```
docker run --rm hello:latest exec hello -t
```

Which should print:

```
hello, world
```

1.2 Install slices in a rock

In this tutorial, you will create a lean hello-world rock that uses chisel slices, and then compare the resulting rock with the one created without slices in *Create a "Hello World" rock*.

1.2.1 Prerequisites

- snap enabled system (https://snapcraft.io)
- LXD installed (https://linuxcontainers.org/lxd/getting-started-cli/)
- skopeo installed (https://github.com/containers/skopeo)
- Docker installed (https://docs.docker.com/get-docker/)
- a text editor

1.2.2 Install Rockcraft

Install Rockcraft on your host:

```
snap install rockcraft --classic
```

1.2.3 Project Setup

Create a new directory, write the following into a text editor and save it as `rockcraft.yaml`:

```
name: chiselled-hello
summary: Hello world from Chisel slices
description: A "bare" rock containing the "hello" package binaries from Chisel slices.
license: Apache-2.0

version: "latest"
base: bare
build_base: "ubuntu@22.04"
platforms:
```

(continues on next page)

(continued from previous page)

```
  amd64:

parts:
  hello:
    plugin: nil
    stage-packages:
      - hello_bins
```

Note that this Rockcraft file uses the `hello_bins` Chisel slice to generate an image containing only files that are strictly necessary for the `hello` binary. See *Chisel* for details on the Chisel tool.

1.2.4 Pack the rock with Rockcraft

To build the rock, run:

```
rockcraft pack
```

The output will look similar to:

```
Launching instance...
Retrieved base bare for amd64
Extracted bare:latest
Executed: pull hello
Executed: overlay hello
Executed: build hello
Executed: stage hello
Executed: prime hello
Executed parts lifecycle
Exported to OCI archive 'chiselled-hello_latest_amd64.rock'
```

The process might take a little while, but at the end, a new file named `chiselled-hello_latest_amd64.rock` will be present in the current directory. That's your chiselled-hello rock, in oci-archive format.

1.2.5 Run the rock in Docker

First, import the recently created rock into Docker:

```
sudo /snap/rockcraft/current/bin/skopeo --insecure-policy copy oci-archive:chiselled-
↪hello_latest_amd64.rock docker-daemon:chiselled-hello:latest
```

Now you can run a container from the rock:

```
docker run --rm chiselled-hello:latest exec hello -t
```

Which should print:

```
hello, world
```

The `chiselled-hello` image will have a size of 5.6 MB, which is much less in size than the 8.8 MB `hello` rock created in *Create a "Hello World" rock*.

The next tutorials walk you through examples of transforming applications source code into container applications:

1.3 Containerise a PyPI package

By the end of this tutorial you will be able to run pyfiglet via docker:

```
$ docker run --rm -it pyfiglet:0.7.6 exec pyfiglet hello
 _          _ _
| |__   ___| | | ___
| '_ \ / _ \ | |/ _ \
| | | |  __/ | | (_) |
|_| |_|\___|_|_|\___/
```

1.3.1 Prerequisites

- snap enabled system (https://snapcraft.io)
- LXD installed (https://linuxcontainers.org/lxd/getting-started-cli/)
- skopeo installed (https://github.com/containers/skopeo). Skopeo will also be automatically installed as a Rockcraft dependency
- Docker installed (https://snapcraft.io/docker)
- Rockcraft installed
- a text editor

1.3.2 Project Setup

To create a new Rockcraft project, create a new directory and change into it:

```
mkdir pyfiglet-rock && cd pyfiglet-rock
```

Next, create a file called `rockcraft.yaml` with the following contents:

```
name: pyfiglet
base: ubuntu@22.04
version: '0.7.6'  # Note: should match `pyfiglet` below
summary: A rock for pyfiglet
description: A rock for pyfiglet
license: Apache-2.0
platforms:
  amd64:

parts:
  pyfiglet:
    plugin: python
    source: .
    python-packages:
      - pyfiglet==0.7.6  # Note: should match `version` above
    stage-packages:
      - python3-venv
```

1.3.3 Pack the rock with Rockcraft

To build the rock, run:

```
rockcraft pack
```

1.3.4 Run the rock in Docker

First, import the recently created rock into Docker:

```
sudo /snap/rockcraft/current/bin/skopeo --insecure-policy copy oci-archive:pyfiglet_0.7.
↪6_amd64.rock docker-daemon:pyfiglet:0.7.6
```

Now run the `pyfiglet` command from the rock:

```
docker run --rm pyfiglet:0.7.6 exec pyfiglet it works!
```

Which should print:

```
 _ _                          _        _
(_) |_  __      _____  _ __| | _____| |
| | __| \ \ /\ / / _ \| '__| |/ / __| |
| | |_   \ V  V / (_) | |  |   <\__ \_|
|_|\__|   \_/\_/ \___/|_|  |_|\_\___(_)
```

1.3.5 Explore the running container

Since the rock uses an ubuntu base, you can poke around in a running container using bash, via:

```
$ docker run --rm -it pyfiglet:0.7.6 exec bash
root@14d1812a2681:/# pyfiglet hi
 _     _
| |__ (_)
| '_ \| |
| | | | |
|_| |_|_|
```

1.4 Bundle a Node.js app within a rock

This tutorial describes the steps needed to bundle a typical Node.js application into a rock.

1.4.1 Prerequisites

- snap enabled system (https://snapcraft.io/docs/installing-snapd)
- LXD installed (https://documentation.ubuntu.com/lxd/en/latest/installing/)
- Docker installed (https://snapcraft.io/docker)
- a text editor

1.4.2 Install Rockcraft

Install Rockcraft on your host:

```
sudo snap install rockcraft --classic
```

1.4.3 Project Setup

Starting in an empty folder, create a `src/` subdirectory. Inside it, add two files:

The first one is the `package.json` listing of dependencies, with the following contents:

Listing 1: package.json

```
{
    "name": "node_web_app",
    "version": "1.0.0",
    "description": "Node.js on a rock",
    "author": "First Last <first.last@example.com>",
    "main": "server.js",
    "scripts": {
      "start": "node server.js"
    },
    "dependencies": {
      "express": "^4.18.2"
    }
}
```

The second file is our sample app, a simple "hello world" server. Still inside `src/`, add the following contents to `server.js`:

Listing 2: server.js

```
'use strict';

const express = require('express')
const app = express()
const port = 8080
const host = '0.0.0.0'

app.get('/', (req, res) => {
  res.send('Hello World from inside the rock!');
});
```

(continues on next page)

(continued from previous page)

```
app.listen(port, host, () => {
  console.log(`Running on http://${host}:${port}`);
});
```

Next, we'll setup the Rockcraft project. In the original empty folder, create an empty file called `rockcraft.yaml`. Then add the following snippets, one after the other:

Add the metadata that describes your rock, such as its name and licence:

Listing 3: rockcraft.yaml

```
name: my-node-app
base: ubuntu@22.04
version: '1.0'
summary: A rock that bundles a simple nodejs app
description: |
  This rock bundles a recent node runtime to serve a simple "hello-world" app.
license: GPL-3.0
platforms:
  amd64:
```

Add the container entrypoint, as a Pebble service:

Listing 4: rockcraft.yaml

```
services:
    app:
        override: replace
        command: node server.js
        startup: enabled
        on-success: shutdown
        on-failure: shutdown
        working-dir: /lib/node_modules/node_web_app
```

Finally, add a part that describes how to build the app created in the `src/` directory using the `npm` plugin:

Listing 5: rockcraft.yaml

```
parts:
    app:
        plugin: npm
        npm-include-node: True
        npm-node-version: "21.1.0"
        source: src/
```

The whole file then looks like this:

Listing 6: rockcraft.yaml

```
name: my-node-app
base: ubuntu@22.04
version: '1.0'
summary: A rock that bundles a simple nodejs app
description: |
```

(continues on next page)

(continued from previous page)

```
  This rock bundles a recent node runtime to serve a simple "hello-world" app.
license: GPL-3.0
platforms:
  amd64:

services:
    app:
        override: replace
        command: node server.js
        startup: enabled
        on-success: shutdown
        on-failure: shutdown
        working-dir: /lib/node_modules/node_web_app

parts:
    app:
        plugin: npm
        npm-include-node: True
        npm-node-version: "21.1.0"
        source: src/
```

1.4.4 Pack the rock with Rockcraft

To build the rock, run:

```
rockcraft pack
```

At the end of the process, a new rock file should be present in the current directory:

```
ls my-node-app_1.0_amd64.rock
```

1.4.5 Run the rock in Docker

First, import the recently created rock into Docker:

```
sudo /snap/rockcraft/current/bin/skopeo --insecure-policy copy oci-archive:my-node-app_1.
↪0_amd64.rock docker-daemon:my-node-app:1.0
```

Since the rock bundles a web-app, we'll first start serving that app on local port 8000:

```
docker run --name my-node-app -p 8000:8080 my-node-app:1.0
```

The output will look similar to this, indicating that Pebble started the app service:

```
2023-10-30T12:37:33.654Z [pebble] Started daemon.
2023-10-30T12:37:33.659Z [pebble] POST /v1/services 3.878846ms 202
2023-10-30T12:37:33.659Z [pebble] Started default services with change 1.
2023-10-30T12:37:33.663Z [pebble] Service "app" starting: node server.js
2023-10-30T12:37:33.864Z [app] Running on http://0.0.0.0:8080
```

Next, open your web browser and navigate to `http://localhost:8000`. You should see a blank page with a "Hello World from inside the rock!" message. Success!

You can now stop the running container by either interrupting it with CTRL+C or by running the following in another terminal:

```
docker stop my-node-app
```

1.4.6 References

The sample app code comes from the "Hello world example" Express tutorial, available at https://expressjs.com/en/starter/hello-world.html.

CHAPTER

TWO

HOW-TO GUIDES

If you have a specific goal but are already familiar with Rockcraft, our How-to guides have more in-depth detail than our tutorials and can be applied to a broader set of applications.

They'll help you achieve an end result but may require you to understand and adapt the steps to fit your specific requirements.

2.1 How to get started - quick guide

See the *Tutorials* for a full getting started guide.

2.1.1 Getting started

Rockcraft is **the tool** for building Ubuntu-based and production-grade OCI images, aka rocks!

Rockcraft is distributed as a snap. For packing new rocks, it makes use of "providers" to execute all the steps involved in the rock's build process. At the moment, the supported providers are LXD and Multipass.

Requirements

Before installing the Rockcraft snap, make sure you have the necessary tools and environment to install and run Rockcraft.

First things first, if you are running Ubuntu, Snap is already installed and ready to go:

```
snap --version
```

You'll get something like:

```
snap    2.57.1
snapd   2.57.1
series  16
ubuntu  22.04
kernel  5.17.0-1016-oem
```

If this is not the case, then please check https://snapcraft.io/docs/installing-snap-on-ubuntu.

For what concerns providers, LXD is the default one for Rockcraft, so start by checking if it is available:

```
lxd --version
```

The output will be something like:

```
5.5
```

And that it is enabled:

```
systemctl status snap.lxd.daemon.service
```

The output should look like:

```
snap.lxd.daemon.service - Service for snap application lxd.daemon
   Loaded: loaded (/etc/systemd/system/snap.lxd.daemon.service; static)
   Active: active (running) since Wed 2022-09-07 16:02:29 CEST; 6 days ago
   ...
```

If LXD is not installed, then run:

```
snap install lxd
```

And if LXD is not running, try starting it via:

```
lxd init --minimal  # drop the --minimal for an interactive configuration
```

May you find any problems with LXD, please check https://ubuntu.com/lxd.

Choose a Rockcraft release

Pick a Rockcraft release, either from the snap store or via `snap search rockcraft`.

Keep in mind the chosen channel, as riskier releases are more prone to breaking changes.

Also, note that the Rockcraft's snap confinement is set to "classic" (this is important for the installation step).

Installation steps

Having chosen a Rockcraft release, you must now install it via the snap CLI (or directly via the Ubuntu Desktop store):

```
sudo snap install rockcraft --channel=<chosen channel> --classic
```

For example:

```
sudo snap install rockcraft --classic
```

Testing Rockcraft

Once installed, you can make sure that Rockcraft is actually present in the system and ready to be used:

```
rockcraft --version
```

The output will be similar to:

```
rockcraft 0.0.1.dev1
```

2.2 Chisel

Upgrade your Chisel slicing skills by learning how to cut, install and publish slices.

2.2.1 Cut existing slices with Chisel

Chisel has been integrated with Rockcraft in a way that it becomes seamless to users. Packages and slices can be both installed via the `stage-packages` field without any ambiguities because slices follow an underscore-driven naming convention. For instance, `openssl` means the whole OpenSSL package, while `openssl_bins` means just the binaries slice of the OpenSSL package. Rockcraft will take care of the installation and priming of your content into the rock. There's an example *here*.

Chisel isn't, however, specific to Rockcraft. It can be used on its own! It relies on a database of slices that are indexed per Ubuntu release. So for example, the following command:

```
chisel cut --release ubuntu-22.04 --root myrootfs libgcc-s1_libs libssl3_libs
```

would look into the Ubuntu Jammy archives, fetch the provided packages and install only the desired slices into the `myrootfs` folder.

To learn more about Chisel and how it works, have a look at the source code.

Do you need a package slice that doesn't exist yet? Please feel free to propose your slice definition for inclusion in Chisel releases.

2.2.2 How to create a package slice for Chisel

If your package doesn't yet have the slice definitions you actually need to **create your own slice definition** (which you can, later on, propose to be merged upstream for everyone else to use *How to publish a slice definition*).

Let's assume you are trying to create a slice definition for installing the OpenSSL binary into your rock!

Make sure the slice definition doesn't exist yet

To avoid re-creating a slice, check the following to see if something that fits your needs already exists:

1. Look into the upstream chisel-releases repository
2. Switch to the branch corresponding to the desired Ubuntu release for your rock
3. Search your package name within the list of slice definitions files
    - if you find it, open it and try to find a slice name containing the bits and pieces you need from that package

Structure of a slice definitions file

There can be only **one slice definitions file** for each Ubuntu package. All of the slice definitions files follow the same structure:

```
# (req) Name of the package.
# The slice definition file should be named accordingly (eg. "openssl.yaml")

package: <package-name>
```

(continues on next page)

(continued from previous page)

```
# (req) List of slices
slices:

    # (req) Name of the slice
    <slice-name>:

        # (opt) Optional list of slices that this slice depends on
        essential:
          - <pkgA_slice-name>
          - ...

        # (req) The list of files, from the package, that this slice will install
        contents:
            </path/to/content>:
            </path/to/another/multiple*/content/**>:
            </path/to/moved/content>: {copy: /bin/original}
            </path/to/link>: {symlink: /bin/mybin}
            </path/to/new/dir>: {make: true}
            </path/to/file/with/text>: {text: "Some text"}
            </path/to/mutable/file/with/default/text>: {text: FIXME, mutable: true}
            </path/to/temporary/content>: {until: mutate}

        # (opt) Mutation scripts, to allow for the reproduction of maintainer scripts,
        # based on Starlark (https://github.com/canonical/starlark)
        mutate: |
            ...
```

Find the dependencies of your package

Find the dependencies of the package for which you want to create a new slice definition (`openssl` in this guide) with this command:

```
apt show openssl
```

The output will be similar to:

```
package: openssl
Version: 3.0.2-0ubuntu1.7
Origin: Ubuntu
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Original-Maintainer: Debian OpenSSL Team <pkg-openssl-devel@alioth-lists.debian.net>
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Depends: libc6 (>= 2.34), libssl3 (>= 3.0.2-0ubuntu1.2)
```

From the above output, you can confirm that `openssl` **depends on** `libc6` and `libssl3`. So when creating your slice definitions file for OpenSSL, you will need to remember to include those packages' slices as a dependency as well, whenever needed. Let's do that in the following section.

Create your slice definition

You now have everything needed to actually define the OpenSSL slice that will install the content you are looking to have in the final rock. Since you are looking to install just the OpenSSL binaries from the `openssl` package, what about naming this new slice **bins**? Let's go for it:

1. **What is the name of your slice definitions file?** It is a YAML file called `openssl.yaml`
2. **What package name should be defined inside this file?** The package name is `openssl`
3. **What is your slice name?** It should be called `bins`
4. **What contents do you need from the OpenSSL package?** Just the binaries - `/usr/bin/c_rehash` and `/usr/bin/openssl`
5. **Does your slice depend on any other package slice?** Yes, OpenSSL depends on `libc6` and `libssl3`
   - **Do these two packages have slice definitions files upstream?** Yes, there is already a slice definitions file for libc6 and another one for libssl3. If these dependencies were not present in the upstream Chisel release, you would also need to create their corresponding slice definitions
   - **Which slices do you depend on then?** Since you only want the OpenSSL binaries, you might only need the libraries from `libc6` and `libssl3`, as well as the configuration files from `libc6` and `openssl` themselves.

Create a new YAML file named `openssl.yaml`, with the following content:

```
package: openssl
slices:
    bins:
        essential:
          - libc6_libs
          - libc6_config
          - libssl3_libs
          - openssl_config
        contents:
            /usr/bin/c_rehash:
            /usr/bin/openssl:

    config:
        contents:
            /etc/ssl/private/:
            /etc/ssl/openssl.cnf:
            /usr/lib/ssl/certs:
            /usr/lib/ssl/openssl.cnf:
            /usr/lib/ssl/private:
```

Notice the unforeseen new slice `config`. Because your OpenSSL binaries depend on the OpenSSL configuration files, and those were not yet present anywhere in the Chisel releases upstream, you also need to create that slice! You may also ask **"why not put those configuration files inside the "bins" slice"**? You could! But we recommend, as a best practice, to separate and group contents according to their nature, as you may tomorrow need to create a new slice definition that only needs the OpenSSL configurations and not the binaries.

And that's it. This is your brand new slice definitions file, which will allow Chisel to install **just** the OpenSSL binaries (and their dependencies) into your rock! To learn about how to actually use this new slice definition file and publish it upstream for others to use, please check the following guides.

2.2.3 How to install a custom package slice

When a specific package slice is not available on the upstream Chisel releases, you will more likely end up creating your own slice definition.

Once you have it though, the most obvious question is: **how can I install this custom slice with Chisel?**

Let's assume you want to install the OpenSSL binaries slice created in *this guide*.

First, clone the Chisel releases repository:

```
# Let's assume we are working with Ubuntu 22.04
git clone -b ubuntu-22.04 https://github.com/canonical/chisel-releases/
```

This repository acts as the database of slice definitions files for each Chisel release (Chisel releases are named analogously to Ubuntu releases, and mapped into Git branches within the repository).

Chisel will only recognise slices belonging to a Chisel release, so you need to copy your slice definitions file - `openssl.yaml` in this example - into the `chisel-releases/slices` folder. Note that if a slice definitions file with the same name already exists, it most likely means that the package you're slicing has already been sliced before, and in this case, you only need to merge your changes into that existing file.

At this point, you should be able to find your custom OpenSSL slice `bins` in the local Chisel release:

```
grep -q "bins" chisel-releases/slices/openssl.yaml && echo "My slice exists"
```

If you wanted to test it with Chisel alone, you could now simply run

```
# Testing with Chisel directly:
mkdir -p my-custom-openssl-fs
chisel cut --release ./chisel-releases --root my-custom-openssl-fs openssl_bins
```

You should end up with a folder named "my-custom-openssl-fs" containing a few folders, amongst which there would be `./usr/bin/openssl`.

To install the custom package slice into a rock though, you need to use Rockcraft!

Start by initialising a new Rockcraft project:

```
rockcraft init
```

After this command, you should find a new `rockcraft.yaml` file in your current path.

Adjust the `rockcraft.yaml` file according to the following content (feel free to adjust the metadata, but pay special attention to the `parts` section):

```
name: custom-openssl-rock
base: bare
build_base: "ubuntu@22.04"
version: '0.0.1'
summary: A chiselled rock with a custom OpenSSL slice
description: |
    A rock containing only the binaries (and corresponding dependencies) from the␣
↪OpenSSL package.
    Built from a custom Chisel release.
license: GPL-3.0
platforms:
    amd64:
```

(continues on next page)

(continued from previous page)

```
parts:
    build-context:
        plugin: nil
        source: chisel-releases/
        source-type: local
        override-build:
            chisel cut --release ./ --root ${CRAFT_PART_INSTALL} openssl_bins
```

The "build-context" part allows you to send the local `chisel-releases` folder into the builder. The "override-build" enables you to install your custom slice. Please not that this level of customisation is only needed when you want to install from a custom Chisel release. If the desired slice definitions are already upstream, then you can simply use `stage-packages`, as demonstrated in *here*.

Build your rock with:

```
rockcraft pack
```

The output will be:

```
Launching instance...
Retrieved base bare for amd64
Extracted bare:latest
Executed: pull build-context
Executed: pull pebble
Executed: overlay build-context
Executed: overlay pebble
Executed: build build-context
Executed: build pebble
Executed: stage build-context
Executed: stage pebble
Executed: prime build-context
Executed: prime pebble
Executed parts lifecycle
Exported to OCI archive 'custom-openssl-rock_0.0.1_amd64.rock'
```

Test that the OpenSSL binaries have been correctly installed with the following:

```
sudo /snap/rockcraft/current/bin/skopeo --insecure-policy copy oci-archive:custom-
↪openssl-rock_0.0.1_amd64.rock docker-daemon:chisel-openssl:latest
```

The output will be:

```
Getting image source signatures
Copying blob 253d707d7e97 done
Copying blob 7044a53e1935 done
Copying config c114b59704 done
Writing manifest to image destination
Storing signatures
```

And after:

```
docker run --rm chisel-openssl exec openssl
```

The output of the Docker command will be OpenSSL's default help message:

```
help:

Standard commands
asn1parse         ca                ciphers           cmp
cms               crl               crl2pkcs7         dgst
dhparam           dsa               dsaparam          ec
ecparam           enc               engine            errstr
fipsinstall       gendsa            genpkey           genrsa
help              info              kdf               list
mac               nseq              ocsp              passwd
pkcs12            pkcs7             pkcs8             pkey
pkeyparam         pkeyutl           prime             rand
rehash            req               rsa               rsautl
s_client          s_server          s_time            sess_id
<... many more lines of output>
```

And that's it! You've now built your own rock from a custom Chisel release. Next step: share your slice definitions file with others!

2.2.4 How to publish a slice definition

At this stage, you have created some package slice definitions and you have a custom Chisel release in your local development environment. You have also tested this custom Chisel release, and it works! You believe there are others who could really use it as well, so **how can you make it accessible to everyone**?

It is as simple as proposing your changes into the upstream Chisel releases repository:

1. Fork this repository https://github.com/canonical/chisel-releases and clone your fork:

```
# Let's assume we are working with Ubuntu 22.04
git clone -b ubuntu-22.04 https://github.com/<your_github_username>/chisel-releases.git
```

2. Create a branch:

```
cd chisel-releases
git checkout -b create-openssl-bins-slice
```

3. Add and commit your modifications:

```
cp <path/to/your/slice/definitions/file> slices
git add slices/
git commit -m "feat: add new slice definitions for 'name_of_the_package'"
git push origin create-openssl-bins-slice
```

Create a pull request and wait for it to be merged.

And that's it! Your custom Chisel release and new slice definitions are now available in Chisel, and anyone can use them. **Congrats**! And thank you for your contribution.

2.3 Documentation

If you have found something missing in this documentation, or have useful information to add whether it is a tutorial, a guide or something else, we compiled a couple of guides for you to facilitate your contribution.

2.3.1 How to contribute to Rockcraft documentation

Tools and markup

Rockcraft's documentation is generated with Sphinx, using reStructuredText as the markup language for the source files.

Documentation structure

The Rockcraft documentation is organised according to the Diátaxis framework. Additionally, some rules are used to ensure that code referred to by the documentation is kept up-to-date and tested thoroughly.

Including code and commands

All code-like content going into the documentation must be tested, especially if it is supposed to be reproducible. Pages that include code snippets or terminal commands need to provide this information in separate files that can be included in the project's test infrastructure.

Categories of the documentation that use code snippets will each have their own directory with a `code` folder within. For example, `tutorials/code` will hold code for the tutorials.

Each page that provides technical information to the reader in the form of code snippets or commands, such as a tutorial or how-to guide, should have its own folder within the `code` directory and contain a `task.yaml` file for spread tests. For example, the `tutorials/hello-world.rst` tutorial refers to code supplied in the `tutorials/code/hello-world/task.yaml` file. Additional YAML files can also be included when needed, as shown here:

```
tutorials
├─ hello-world.rst
└─ code
   └─ hello-world
      ├─ task.yaml
      └─ rockcraft.yaml
```

These YAML files can also include more than just the snippets that appear in a page. For example, they can include additional commands to set up a test environment or clean up after the test has been run. Each snippet should be delimited with comments that enable them to be conveniently extracted, as in this example:

```
  # [docs:snap-version]
  snap --version
  # [docs:snap-version-end]
```

When including code snippets in a page, use the reStructuredText literalinclude directive with the `start-after` and `end-before` options to extract the relevant lines of text. If the indentation of the quoted code is excessive, use the `dedent` option to reduce it to an acceptable level, as in this example:

```
.. literalinclude:: code/get-started/task.yaml
    :language: bash
    :start-after: [docs:snap-version]
    :end-before: [docs:snap-version-end]
    :dedent: 2
```

Handling code snippets in this systematic way encourages reuse of existing code snippets and helps to ensure that the documentation stays up-to-date.

Build the documentation

The *How to build the documentation* guide contains step-by-step instructions for setting up a virtual environment and building the documentation.

Making a contribution

Report issues

If you have a request or found a problem with the documentation, then please submit an issue. These issues are supervised and regularly triaged by the repository owners. If you don't receive an answer within 2 weeks, please reply to your own issue asking for an update.

Propose changes

Community contributions are more than welcome. To become an external contributor you should:

1. create a fork of the Rockcraft repository,
2. **commit the changes to your fork (ideally to a new branch),**
    1. make sure to follow the project's CONTRIBUTING guidelines,
3. create a Pull Request against the `main` branch.

Similarly to new issues, new Pull Requests (PR) are also supervised and regularly triaged by the repository owners. If the tests are passing and you don't receive an answer within 2 weeks, please add a comment to your own PR asking for an update.

2.3.2 How to build the documentation

Use the provided `Makefile` to install the documentation requirements:

```
make installdocs
```

Once the requirements are installed, you can use the provided `Makefile` to build the documentation:

```
make docs  # the home page can be found at docs/_build/html/index.html
```

Even better, serve it locally on port 8080. The documentation will be rebuilt on each file change, and will reload the browser view.

```
make rundocs
```

Note that `make rundocs` automatically activates the virtual environment, as long as it already exists.

2.4 Rocks

Learn about how to pack a rock, how to migrate a Dockerfile to a rock, or how to use services in your rockcraft.yaml. Those guides will help you explore all those aspects in Rockcraft and more.

2.4.1 How to use Rockcraft's GitHub Action

Within your GitHub repository, make sure you have GitHub Actions enabled.

Navigate to `.github/workflows`, open the YAML file where you want the rock build to take place, and add the following steps:

```
steps:
  - uses: actions/checkout@v3
  - uses: canonical/craft-actions/rockcraft-pack@main
```

Commit and push the changes. This will trigger a new workflow run using Rockcraft to pack your rock based on the `rockcraft.yaml` file at your project's root.

To learn how to publish this rock outside the GitHub build environment and how to pass additional input parameters to this action, please refer to the action's documentation.

2.4.2 How to convert an entrypoint to a Pebble layer

This guide will show you how to take an existing Docker image entrypoint and convert it into a Pebble layer, aka the list of one or more services which is defined in `rockcraft.yaml` and then taken by the rock's Pebble entrypoint.

Reference entrypoint

For this guide, the reference Docker image entrypoint will be NGINX. The official Debian-based NGINX image's Dockerfile can be found here.

In summary, this Dockerfile is basically installing NGINX into the image and then defining the OCI entrypoint to be a custom shell script which parses the first argument given to it at container deployment time, and then configures and launches NGINX accordingly.

Design the Pebble services

A Pebble layer is composed of metadata, checks and services. The latter is present in `rockcraft.yaml` as a top-level field and it represents the services which are loaded by the Pebble entrypoint when deploying a rock.

Given the reference entrypoint, this guide's goal is to create two services: one for `nginx` and another for `nginx-debug`. The following `services` snippet does just that:

```
services:
  nginx:
    override: replace
    startup: disabled
    command: nginx [ -h ]
```

(continues on next page)

(continued from previous page)

```
    environment:
      TZ: UTC
    on-failure: shutdown
  nginx-debug:
    override: replace
    startup: disabled
    command: nginx-debug [ -h ]
    environment:
      TZ: UTC
    on-failure: shutdown
```

This is defining two separate Pebble services which are disabled by default at startup, have the same environment variable, but are executed with different commands (`nginx` and `nginx-debug`).

Build the rock

Copy the above snippet and incorporate it into the `rockcraft.yaml` file which will be used to build your rock, as shown below:

```
name: custom-nginx-rock
base: "ubuntu@22.04"
version: latest
summary: An NGINX rock
description: |
    A rock equivalent of the official NGINX Docker image from Docker Hub.
license: Apache-2.0
platforms:
    amd64:

package-repositories:
  - type: apt
    url: https://nginx.org/packages/mainline/ubuntu
    key-id: 573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62
    suites:
      - jammy
    components:
      - nginx
    priority: always

parts:
  nginx-user:
    plugin: nil
    overlay-script: |
      set -x
      useradd -R $CRAFT_OVERLAY -M -U -r nginx

  nginx:
    plugin: nil
    after:
      - nginx-user
```

(continues on next page)

(continued from previous page)

```
    stage-packages:
      - nginx
      - tzdata

# Services to be loaded by the Pebble entrypoint
services:
  nginx:
    override: replace
    startup: disabled
    command: nginx [ -h ]
    environment:
      TZ: UTC
    on-failure: shutdown
  nginx-debug:
    override: replace
    startup: disabled
    command: nginx-debug [ -h ]
    environment:
      TZ: UTC
    on-failure: shutdown
```

This Rockcraft recipe is fully declarative, with the creation of the "nginx" user being the only scripted step.

To reproduce what the reference NGINX Dockerfile is doing, notice the use of `package-repositories` in this `rockcraft.yaml` file, allowing you to also make use of NGINX's 3rd party package repository (even using the same GPG key ID as the one used in the Dockerfile).

NOTE: to add custom configuration files, you can use the `dump` plugin.

Now, build the final custom NGINX rock with:

```
rockcraft pack
```

You should see something like this:

```
Launching instance...
Retrieved base ubuntu@22.04 for amd64
Extracted ubuntu@22.04
Refreshing repositories | (4.6s)
Package repositories installed
Executed: pull nginx-user
Executed: pull nginx
Executed: pull pebble
Executed: overlay nginx-user
Executed: overlay nginx
Executed: overlay pebble
Executed: build nginx-user
Executed: skip pull nginx-user (already ran)
Executed: skip overlay nginx-user (already ran)
Executed: skip build nginx-user (already ran)
Executed: stage nginx-user (required to build 'nginx')
Executed: build nginx
Executed: build pebble
```

(continues on next page)

(continued from previous page)

```
Executed: skip stage nginx-user (already ran)
Executed: stage nginx
Executed: stage pebble
Executed: prime nginx-user
Executed: prime nginx
Executed: prime pebble
Executed parts lifecycle
Exported to OCI archive 'custom-nginx-rock_latest_amd64.rock'
```

Then copy the resulting rock (from the OCI archive format) to the Docker daemon via:

```
sudo /snap/rockcraft/current/bin/skopeo --insecure-policy copy oci-archive:custom-nginx-
↪rock_latest_amd64.rock docker-daemon:custom-nginx-rock:latest
```

And finally, run the container:

```
docker run -d --name nginx-pebble-service -p 8080:80 custom-nginx-rock:latest --args␣
↪nginx -g 'daemon off;' \; start nginx
```

Notice the given command `start nginx`, as this is Pebble's client syntax to let the Pebble daemon know that the `nginx` service defined in `rockcraft.yaml` (which is disabled by default) should be enabled at startup. Otherwise, the Pebble daemon would start without any NGINX service, although you could still later on ask for that service to be started (via something like `docker exec <container-name> start nginx`).

At this point, your container should be deployed and running the `nginx` service, and you should be able to see the NGINX landing page by accessing port 8080 on you localhost:

```
curl localhost:8080
```

For which you should see the following output:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

2.4.3 Publish a rock to a registry

Prerequisites

- skopeo installed (https://github.com/containers/skopeo)
- Docker installed (https://docs.docker.com/get-docker/)

Push a rock to Docker Hub

The output of `rockcraft pack` is a rock in its oci-archive archive format.

```
skopeo --insecure-policy copy oci-archive:<your_rock_file.rock> docker://<container_
↪registry>/<repo>:<tag>
```

Output:

```
Getting image source signatures
Copying blob e65b2e587073 skipped: already exists
Copying blob 01f981dde5a5 skipped: already exists
Copying config 5da22a9016 done
Writing manifest to image destination
Storing signatures
```

2.4.4 Migrate a popular Docker image to a chiselled rock

Prerequisites

- snap enabled system (https://snapcraft.io)
- LXD installed (https://linuxcontainers.org/lxd/getting-started-cli/)
- skopeo installed (https://github.com/containers/skopeo). Skopeo will also be automatically installed as a Rockcraft dependency
- Docker installed (https://snapcraft.io/docker)
- a text editor

Install Rockcraft

Install Rockcraft on your host:

```
sudo snap install rockcraft --classic
```

Project Setup

For this tutorial, the reference Docker image will be Microsoft's .NET Runtime 6.0. The target base Ubuntu release will be Jammy, and the target architecture will be `amd64`.

Create a new directory, write the reference Dockerfile (pasted below) into a text editor and save it as `Dockerfile`:

```
ARG REPO=mcr.microsoft.com/dotnet/runtime-deps

# Installer image
FROM amd64/buildpack-deps:jammy-curl AS installer

# Retrieve .NET Runtime
RUN dotnet_version=6.0.16 \
    && curl -fSL --output dotnet.tar.gz https://dotnetcli.azureedge.net/dotnet/Runtime/
↪$dotnet_version/dotnet-runtime-$dotnet_version-linux-x64.tar.gz \
    && dotnet_sha512=
↪'c8891b791a51e7d2c3164470dfd2af2ce59af3c26404e84075277e307df7dcd1e3ccf1a1a3c2655fe2eea8a30f8349b7adbb
↪' \
    && echo "$dotnet_sha512  dotnet.tar.gz" | sha512sum -c - \
    && mkdir -p /dotnet \
    && tar -oxzf dotnet.tar.gz -C /dotnet \
    && rm dotnet.tar.gz

# .NET runtime image
FROM $REPO:6.0.24-jammy-amd64

# .NET Runtime version
ENV DOTNET_VERSION=6.0.16

COPY --from=installer ["/dotnet", "/usr/share/dotnet"]

RUN ln -s /usr/share/dotnet/dotnet /usr/bin/dotnet
```

For the sake of comparison, start by building this Docker image by running:

```
docker build -t dotnet-runtime:reference .
```

The output should look as follows:

```
[+] Building 0.6s (10/10) FINISHED
=> [internal] load .dockerignore0.0s
=> => transferring context: 2B0.0s
=> [internal] load build definition from Dockerfile    0.0s
=> => transferring dockerfile: 881B   0.0s
=> [internal] load metadata for mcr.microsoft.com/dotnet/runtime-deps:6.0.16-jammy-amd64␣
↪   0.1s
=> [internal] load metadata for docker.io/amd64/buildpack-deps:jammy-curl    0.6s
=> [stage-1 1/3] FROM mcr.microsoft.com/dotnet/runtime-deps:6.0.16-jammy-
↪amd64@sha256:e764c6f0cc866a1f2932  0.0s
=> [installer 1/2] FROM docker.io/amd64/buildpack-deps:jammy-
↪curl@sha256:e1f00c6daf4cd328bbef9c52e6c60f18a  0.0s
=> CACHED [installer 2/2] RUN dotnet_version=6.0.16&& curl -fSL --output dotnet.tar.gz␣
↪https://dotnet  0.0s
```

(continues on next page)

(continued from previous page)

```
=> CACHED [stage-1 2/3] COPY --from=installer [/dotnet, /usr/share/dotnet]   0.0s
=> CACHED [stage-1 3/3] RUN ln -s /usr/share/dotnet/dotnet /usr/bin/dotnet   0.0s
=> exporting to image   0.0s
=> => exporting layers  0.0s
=> => writing image␣
↪sha256:a24cab51d4d02019dafcd22a2e2a3e1e6d033f9bbf1cb401d465cb2426bb2264 0.0s
=> => naming to docker.io/library/dotnet-runtime:reference   0.0s
```

Now, inspect this .NET reference image's size:

```
docker images dotnet-runtime:reference
```

The output should look as follows:

```
REPOSITORY TAG IMAGE ID CREATED SIZE
dotnet-runtime   reference   a24cab51d4d0   4 minutes ago   187MB
```

And make sure it is functional:

```
docker run --rm dotnet-runtime:reference dotnet --info
```

The output should look as follows:

```
global.json file:
Not found

Host:
Version:6.0.16
Architecture: x64
Commit: 1e620a42e7

.NET SDKs installed:
No SDKs were found.

.NET runtimes installed:
Microsoft.NETCore.App 6.0.16 [/usr/share/dotnet/shared/Microsoft.NETCore.App]

Download .NET:
https://aka.ms/dotnet-download

Learn about .NET Runtimes and SDKs:
https://aka.ms/dotnet/runtimes-sdk-info
```

Convert Dockerfile to rockcraft.yaml file

From a quick analysis of the reference Dockerfile above, the following requirements must be met:

- R1. The rock must be based on Ubuntu Jammy
- R2. There is no predefined Entrypoint or default command
- R3. The rock must have version 6.0 of the .NET Runtime installed
- R4. `/usr/bin/dotnet` must be a symbolic link to the .NET binary

With these requirements in mind, and in the same directory as the Dockerfile from above, write the following into a text editor and save it as `rockcraft.yaml`:

```
name: dotnet-runtime
summary: .NET Runtime 6.0
description: A Chiselled rock for the .NET Runtime 6.0
version: chiselled

# Use a "bare" base for an even smaller rock
base: bare

# Meeting requirement R1 by making sure the rock builds on Jammy
build-base: ubuntu@22.04

license: Apache-2.0

# Target architecture is amd64
platforms:
  amd64:  # Make sure this value matches your computer's architecture

# For meeting requirement R2, simply don't specify any entrypoint (aka "services")

parts:
  install-dotnet-runtime:
    plugin: nil

    # Based on requirement R3, install version 6.0 of the .NET runtime "libs" slice
    stage-packages:
      - base-files_base
      - dotnet-runtime-6.0_libs

    # Based on requirement R4, create the symbolic link
    override-prime: |
      craftctl default
      ln -s /usr/lib/dotnet/dotnet usr/bin/
```

Note the subtle chiselling of the .NET Runtime package in the `rockcraft.yaml` file above. You are requesting Rockcraft to install the `libs` slice of the `dotnet-runtime` deb, which is defined in the ubuntu-22.04 Chisel release.

Pack the Chiselled Rock with Rockcraft

To build the rock, run:

```
rockcraft pack --verbosity debug
```

The output should be similar to:

```
2023-04-19 15:52:48.045 Starting Rockcraft 0.0.1.dev1
...
2023-04-19 15:52:48.214 Launching instance...
2023-04-19 15:52:48.214 Executing on host: lxc remote list --format=yaml
...
2023-04-19 15:58:25.138 :: 2023-04-19 13:56:57.784 Executing parts lifecycle: build␣
↪install-dotnet-runtime
2023-04-19 15:58:25.138 :: 2023-04-19 13:56:57.784 Executing action
2023-04-19 15:58:25.138 :: 2023-04-19 13:56:57.784 execute action install-dotnet-
↪runtime:Action(part_name='install-dotnet-runtime', step=Step.BUILD, action_
↪type=ActionType.RUN, reason=None, project_vars=None,␣
↪properties=ActionProperties(changed_files=None, changed_dirs=None))
2023-04-19 15:58:25.138 :: 2023-04-19 13:56:57.785 load state file: /root/parts/install-
↪dotnet-runtime/state/pull
2023-04-19 15:58:25.138 :: 2023-04-19 13:56:58.081 :: 2023/04/19 13:56:58 Consulting␣
↪release repository...
2023-04-19 15:58:25.138 :: 2023-04-19 13:56:58.349 :: 2023/04/19 13:56:58 Fetching␣
↪current ubuntu-22.04 release...
2023-04-19 15:58:25.138 :: 2023-04-19 13:56:58.351 :: 2023/04/19 13:56:58 Processing␣
↪ubuntu-22.04 release...
2023-04-19 15:58:25.138 :: 2023-04-19 13:56:58.369 :: 2023/04/19 13:56:58 Selecting␣
↪slices...
2023-04-19 15:58:25.138 :: 2023-04-19 13:56:58.369 :: 2023/04/19 13:56:58 Fetching␣
↪ubuntu 22.04 jammy suite details...
2023-04-19 15:58:25.138 :: 2023-04-19 13:57:04.548 :: 2023/04/19 13:57:04 Release date:␣
↪Thu, 21 Apr 2022 17:16:08 UTC
2023-04-19 15:58:25.138 :: 2023-04-19 13:57:04.549 :: 2023/04/19 13:57:04 Fetching index␣
↪for ubuntu 22.04 jammy main component...
2023-04-19 15:58:25.138 :: 2023-04-19 13:57:05.684 :: 2023/04/19 13:57:05 Fetching index␣
↪for ubuntu 22.04 jammy universe component...
2023-04-19 15:58:25.138 :: 2023-04-19 13:57:25.215 :: 2023/04/19 13:57:25 Fetching␣
↪ubuntu 22.04 jammy-security suite details...
2023-04-19 15:58:25.138 :: 2023-04-19 13:57:25.289 :: 2023/04/19 13:57:25 Release date:␣
↪Wed, 19 Apr 2023 12:55:48 UTC
2023-04-19 15:58:25.139 :: 2023-04-19 13:57:25.289 :: 2023/04/19 13:57:25 Fetching index␣
↪for ubuntu 22.04 jammy-security main component...
2023-04-19 15:58:25.139 :: 2023-04-19 13:57:30.489 :: 2023/04/19 13:57:30 Fetching index␣
↪for ubuntu 22.04 jammy-security universe component...
2023-04-19 15:58:25.139 :: 2023-04-19 13:57:32.522 :: 2023/04/19 13:57:32 Fetching␣
↪ubuntu 22.04 jammy-updates suite details...
2023-04-19 15:58:25.139 :: 2023-04-19 13:57:32.667 :: 2023/04/19 13:57:32 Release date:␣
↪Wed, 19 Apr 2023 13:29:07 UTC
2023-04-19 15:58:25.139 :: 2023-04-19 13:57:32.668 :: 2023/04/19 13:57:32 Fetching index␣
↪for ubuntu 22.04 jammy-updates main component...
2023-04-19 15:58:25.139 :: 2023-04-19 13:57:33.631 :: 2023/04/19 13:57:33 Fetching index␣
↪for ubuntu 22.04 jammy-updates universe component...
```

(continues on next page)

(continued from previous page)

```
2023-04-19 15:58:25.139 :: 2023-04-19 13:57:37.908 :: 2023/04/19 13:57:37 Fetching pool/
↪main/b/base-files/base-files_12ubuntu4.3_amd64.deb...
2023-04-19 15:58:25.139 :: 2023-04-19 13:57:38.157 :: 2023/04/19 13:57:38 Fetching pool/
↪main/g/glibc/libc6_2.35-0ubuntu3.1_amd64.deb...
2023-04-19 15:58:25.139 :: 2023-04-19 13:57:40.834 :: 2023/04/19 13:57:40 Fetching pool/
↪main/g/gcc-12/libgcc-s1_12.1.0-2ubuntu1~22.04_amd64.deb...
2023-04-19 15:58:25.140 :: 2023-04-19 13:57:41.262 :: 2023/04/19 13:57:41 Fetching pool/
↪main/g/gcc-12/libstdc++6_12.1.0-2ubuntu1~22.04_amd64.deb...
2023-04-19 15:58:25.140 :: 2023-04-19 13:57:42.001 :: 2023/04/19 13:57:42 Fetching pool/
↪universe/d/dotnet6/dotnet-host_6.0.116-0ubuntu1~22.04.1_amd64.deb...
2023-04-19 15:58:25.140 :: 2023-04-19 13:57:42.119 :: 2023/04/19 13:57:42 Fetching pool/
↪universe/d/dotnet6/dotnet-hostfxr-6.0_6.0.116-0ubuntu1~22.04.1_amd64.deb...
2023-04-19 15:58:25.140 :: 2023-04-19 13:57:42.237 :: 2023/04/19 13:57:42 Fetching pool/
↪main/i/icu/libicu70_70.1-2_amd64.deb...
2023-04-19 15:58:25.140 :: 2023-04-19 13:57:44.046 :: 2023/04/19 13:57:44 Fetching pool/
↪main/u/ust/liblttng-ust-common1_2.13.1-1ubuntu1_amd64.deb...
2023-04-19 15:58:25.140 :: 2023-04-19 13:57:44.146 :: 2023/04/19 13:57:44 Fetching pool/
↪main/n/numactl/libnuma1_2.0.14-3ubuntu2_amd64.deb...
2023-04-19 15:58:25.141 :: 2023-04-19 13:57:44.247 :: 2023/04/19 13:57:44 Fetching pool/
↪main/u/ust/liblttng-ust-ctl5_2.13.1-1ubuntu1_amd64.deb...
2023-04-19 15:58:25.141 :: 2023-04-19 13:57:44.355 :: 2023/04/19 13:57:44 Fetching pool/
↪main/u/ust/liblttng-ust1_2.13.1-1ubuntu1_amd64.deb...
2023-04-19 15:58:25.141 :: 2023-04-19 13:57:44.571 :: 2023/04/19 13:57:44 Fetching pool/
↪main/o/openssl/libssl3_3.0.2-0ubuntu1.8_amd64.deb...
2023-04-19 15:58:25.141 :: 2023-04-19 13:57:45.111 :: 2023/04/19 13:57:45 Fetching pool/
↪main/l/llvm-toolchain-13/libunwind-13_13.0.1-2ubuntu2.1_amd64.deb...
2023-04-19 15:58:25.141 :: 2023-04-19 13:57:45.213 :: 2023/04/19 13:57:45 Fetching pool/
↪main/x/xz-utils/liblzma5_5.2.5-2ubuntu1_amd64.deb...
2023-04-19 15:58:25.141 :: 2023-04-19 13:57:53.384 :: 2023/04/19 13:57:53 Fetching pool/
↪main/libu/libunwind/libunwind8_1.3.2-2build2_amd64.deb...
2023-04-19 15:58:25.141 :: 2023-04-19 13:57:53.523 :: 2023/04/19 13:57:53 Fetching pool/
↪main/z/zlib/zlib1g_1.2.11.dfsg-2ubuntu9.2_amd64.deb...
2023-04-19 15:58:25.141 :: 2023-04-19 13:57:53.982 :: 2023/04/19 13:57:53 Fetching pool/
↪universe/d/dotnet6/dotnet-runtime-6.0_6.0.116-0ubuntu1~22.04.1_amd64.deb...
...
2023-04-19 15:56:00.566 :: 2023-04-19 13:55:59.756 Created new layer
2023-04-19 15:56:00.566 :: 2023-04-19 13:55:59.758 Adding Pebble entrypoint
2023-04-19 15:56:00.566 :: 2023-04-19 13:55:59.758 Configuring entrypoint...
2023-04-19 15:56:00.567 :: 2023-04-19 13:55:59.767 Entrypoint set to ['/bin/pebble',
↪'enter']
2023-04-19 15:56:00.567 :: 2023-04-19 13:55:59.767 Adding metadata
2023-04-19 15:56:00.567 :: 2023-04-19 13:55:59.767 Configuring labels and annotations...
2023-04-19 15:56:00.567 :: 2023-04-19 13:55:59.785 Labels and annotations set to ['org.
↪opencontainers.image.version=chiselled', 'org.opencontainers.image.title=dotnet-runtime
↪', 'org.opencontainers.image.ref.name=dotnet-runtime', 'org.opencontainers.image.
↪licenses=Apache-2.0', 'org.opencontainers.image.created=2023-04-19T13:55:59.
↪767870+00:00', 'org.opencontainers.image.base.
↪digest=13155b5ad26816d4107ee499de072069a701c9fe183f7e347e8d88fee16e69c2']
2023-04-19 15:56:00.567 :: 2023-04-19 13:55:59.792 Setting the ROCK's Control Data
2023-04-19 15:56:00.567 :: 2023-04-19 13:55:59.797 Adding to layer: /tmp/tmpdqjmducj/.
↪rock as '.rock'
2023-04-19 15:56:00.567 :: 2023-04-19 13:55:59.797 Adding to layer: /tmp/tmpdqjmducj/.
```

(continues on next page)

(continued from previous page)

```
↪rock/metadata.yaml as '.rock/metadata.yaml'
2023-04-19 15:56:00.567 :: 2023-04-19 13:55:59.803 Control data written
2023-04-19 15:56:00.567 :: 2023-04-19 13:55:59.804 Metadata added
2023-04-19 15:56:00.567 :: 2023-04-19 13:55:59.804 Exporting to OCI archive
2023-04-19 15:56:00.567 :: 2023-04-19 13:56:00.148 Exported to OCI archive 'dotnet-
↪runtime_chiselled_amd64.rock'
```

At the end of the process, a file named `dotnet-runtime_chiselled_amd64.rock` should be present in the current directory. That's your chiselled rock, as an OCI archive.

Test the rock

First, import the recently created rock into Docker:

```
sudo /snap/rockcraft/current/bin/skopeo --insecure-policy copy oci-archive:dotnet-
↪runtime_chiselled_amd64.rock docker-daemon:dotnet-runtime:chiselled
```

Now inspect the chiselled .NET Runtime rock the same way as it was done for the reference Docker image:

```
docker images dotnet-runtime:chiselled
```

Which should print something like:

```
REPOSITORY TAG IMAGE ID CREATED SIZE
dotnet-runtime   chiselled   4e0951d180e3   About a minute ago   124MB
```

And make sure this rock is as functional as the reference Docker image:

```
docker run --rm dotnet-runtime:chiselled exec dotnet --info
```

The output should be similar to:

```
global.json file:
Not found

Host:
Version:6.0.16
Architecture: x64
Commit: 1e620a42e7

.NET SDKs installed:
No SDKs were found.

.NET runtimes installed:
Microsoft.NETCore.App 6.0.16 [/usr/lib/dotnet/shared/Microsoft.NETCore.App]

Download .NET:
https://aka.ms/dotnet-download

Learn about .NET Runtimes and SDKs:
https://aka.ms/dotnet/runtimes-sdk-info
```

Conclusion

In this tutorial, you have migrated from an imperative container build recipe (Dockerfile) to a declarative one (`rockcraft.yaml`), without any overhead on the final recipe's size or complexity.

The resulting rock ended up being 63MB smaller than the reference one, while offering the same .NET Runtime functionality.

2.4.5 Use the flask-framework extension

Note: The Flask extension is compatible with the `bare` and `ubuntu@22.04` bases.

To employ it, include `extensions: [ flask-framework ]` in your `rockcraft.yaml` file.

Example:

```
name: example-flask
summary: A Flask application
description: A rock packing a Flask application via the flask extension
version: "0.1"
base: bare
build-base: ubuntu@22.04
license: Apache-2.0

extensions:
  - flask-framework

platforms:
  amd64:
```

2.4.6 Managing project files with the flask extension

By default the flask extension only includes `app.py`, `static/`, `app/`, and `templates/` in the flask project, but you can overwrite this behaviour with a prime declaration in the specially-named `flask-framework/install-app` part to instruct the flask extension on which files to include or exclude from the project directory in the rock image.

The extension places the files from the project folder in the `/flask/app` directory in the final image - therefore, all inclusions and exclusions must be prefixed with `flask/app`.

For example, to include only select files:

```
parts:
  flask-framework/install-app:
    prime:
      - flask/app/static
      - flask/app/.env.production
      - flask/app/app.py
      - flask/app/templates
```

To exclude certain files from the project directory in the rock image, add the following part to `rockcraft.yaml`:

```
parts:
  flask-framework/install-app:
    prime:
      - -flask/app/.git
      - -flask/app/.venv
      - -flask/app/.yarn
      - -flask/app/node_modules
```

2.4.7 Use the django-framework extension

Note: The Django extension is compatible with the `bare` and `ubuntu@22.04` bases.

To use it, include `extensions: [ django-framework ]` in your `rockcraft.yaml` file.

Example:

```
name: example-django
summary: A Django application
description: A rock packing a Django application
version: "0.1"
base: ubuntu@22.04
license: Apache-2.0

extensions:
  - django-framework

platforms:
  amd64:
  arm64:
```

2.4.8 Managing project files with the Django extension

The extension will search for a directory named after the rock within the Rockcraft project directory to transfer it into the rock image. The Django project should have a directory named after the rock, and the `wsgi.py` file within this directory must contain an object named `application` to serve as the WSGI entry point.

The following is a typical Rockcraft project that meets this requirement.

```
+-- example_django
|   |-- example_django
|   |   |-- wsgi.py
|   |   +-- ...
|   |-- manage.py
|   |-- migrate.sh
|   +-- some_app
|       |-- views.py
|       +-- ...
|-- requirements.txt
+-- rockcraft.yaml
```

To override this behaviour and adopt a different project structure, add the `django-framework/install-app` part to install the Django project in the `/django/app` directory within the rock image and update the command for the `django` service to point to the WSGI path of your project.

```
parts:
  django-framework/install-app:
    plugin: dump
    source: .
    organize:
      foobar: django/app/foobar
      manage.py: django/app/manage.py
    stage:
      - django/app/foobar
      - django/app/manage.py
    prime:
      - django/app/foobar
      - django/app/manage.py
```

2.4.9 Including local files and remote resources

Craft-parts provides the built-in *dump plugin* for all kinds of projects that need to include local files and remote resources as is. This plugin uses the *source* property in the part to download, unpack, and copy files and directories from the given source to the build environment, then do some organizing if needed, and include them in the final payload.

If you don't need to copy these files, consider using the `nil` plugin. If you need to download and build them from source, consider using plugins for the corresponding languages, such as the *python plugin* and the *rust plugin*.

The typical use cases for the `dump` plugin:

- Include static files in the final payload, such as scripts, configurations, services, documentation, media files, or other resources that are not generated by the build process.
- Include pre-built third-party packages, libraries, binaries, or other artifacts that are not available in the system's package manager or in the project's source code.

For supported source types please refer to *source-type*.

Example: To include a local directory and move files to the correct locations

Given the following project structure:

```
.
└── misc
    ├── fonts
    │   └── good.otf
    └── services
        ├── README
        └── hello.service
```

The following dump part can be used to include the `hello.service` and `good.otf` files from the `misc` directory, then move them to the correct locations, keeping only `/usr` in the final payload:

```
parts:
  my-part:
```

(continues on next page)

(continued from previous page)

```
    plugin: dump
    source: ./misc
    source-type: local
    organize:
      'services/*.service': usr/lib/systemd/system/
      'fonts/*': usr/share/fonts/
    stage:
      - usr/
```

The resulting payload will look like this:

```
.
└── usr
    ├── lib
    │   └── systemd
    │       └── system
    │           └── hello.service
    └── share
        └── fonts
            └── good.otf
```

Example: To include a remote third-party deb package from a URL

The following dump part downloads a `busybox-static` deb package from a remote URL and includes it in the final payload:

```
parts:
  my-part:
    plugin: dump
    source: http://archive.ubuntu.com/ubuntu/pool/main/b/busybox/busybox-static_1.30.1-
↪7ubuntu3_amd64.deb
    source-type: deb
```

The resulting payload will look like this:

```
.
├── bin
│   ├── busybox
│   └── static-sh -> busybox
└── usr
    └── share
        ├── doc
        │   └── busybox-static
        │       └── ...
        └── man
            └── ...
```

Example: To include a remote third-party pre-compiled archive from a URL

The following dump part downloads a pre-compiled git version of the `ffmpeg` tar archive (xz compressed) from a remote URL and only includes the `ffmpeg` and `ffprobe` binaries in the `/usr/bin`.

```
parts:
  my-part:
    plugin: dump
    source: https://johnvansickle.com/ffmpeg/builds/ffmpeg-git-amd64-static.tar.xz
    source-type: tar
    organize:
      'ffprobe': usr/bin/
      'ffmpeg': usr/bin/
    stage:
      - usr/
```

The resulting payload will look like this:

```
.
└── usr
    └── bin
        ├── ffmpeg
        └── ffprobe
```

Example: To include a remote git repository with a specific branch

The following dump part will clone a theme from a remote git repository and move the theme files to the correct location.

```
parts:
  my-part:
    plugin: dump
    source: https://github.com/snapcore/plymouth-theme-ubuntu-core.git
    source-type: git
    source-branch: main
    source-depth: 1
    organize:
      ubuntu-core: usr/share/plymouth/themes/ubuntu-core
```

The resulting payload will look like this:

```
.
├── README.md
├── copyright
└── usr
    └── share
        └── plymouth
            └── themes
                └── ubuntu-core
                    ├── throbber-1.png
                    └── ...
```

2.4.10 Overriding the default build

Craft-parts provides built-in *plugins* for a number of different programming languages, frameworks, and build tools. Since it's not possible to support *every* possible configuration and scenario for each of these technologies, each plugin emits a series of build commands to reproduce what is most typically done for the given domain; for instance, the `make` plugin generates code that calls `make; make install` at build-time.

For cases where a given project being built does *not* follow the typical path, craft-parts provides a way to declare the build commands for a specific part via the *override-build* keyword.

Typical reasons for using `override-build` include:

- Having to run commands before or after the plugin's default commands;
- Building a project that uses a technology (programming language, framework, or build tool) that is not supported by craft-part's *default plugins*;
- More generally, using the `nil` plugin (which has no default build commands).

Follow these steps to ensure a successful build, and see also a general description of the *Build process*.

Determine that you *do* need to use `override-build`

The default plugins strive to implement the most common build process for a given technology but they typically also provide plugin-specific options that allow some degree of customization. As some examples:

- The `make` plugin exposes the `make-parameters` option to allow passing parameters that might be specific to the project's `Makefile`;
- The `npm` plugin exposes the `npm_node_version` option to select the specific version of `npm` that should be used during the build;
- The `python` plugin exposes the `python-packages` and `python-requirements` options to declare specific packages and requirements files that should be used when creating the build's virtual environment.

See the documentation for the plugins that are relevant to your project to determine whether the default process is suitable for you.

Ensure you place the built artefacts in the correct place

The purpose of the Build step in the lifecycle is to generate software artefacts to be included in the final payload. This is achieved by populating a special "install" directory - the contents of this directory will then move forward to the Stage and Prime lifecycle steps. A very common mistake when overriding a part's Build is failing to place the created artefacts in the correct directory.

The location of the "install" directory is stored in the `${CRAFT_PART_INSTALL}` environment variable. This variable is set by craft-parts' tooling when calling the script contained in `override-build`. Therefore, in many cases the build script can simply call the project's build tool with `${CRAFT_PART_INSTALL}` as the output directory. Some examples:

- Go projects can use either `-o "${CRAFT_PART_INSTALL}"` or set `GOBIN` to `${CRAFT_PART_INSTALL}/bin` when calling `go build` or `go install`. This is in part what the `go` plugin does;
- The `dump` plugin copies the entire `source` to the "install" dir. This is achieved by `cp`'ing the contents of the source directory directly to `${CRAFT_PART_INSTALL}`;
- The `npm` plugin sets the `--prefix` option of `npm install` to `${CRAFT_PART_INSTALL}`;
- The `make` plugin sets the commonly-used `DESTDIR` variable to `${CRAFT_PART_INSTALL}` to ensure that `make install` places the built artefacts in the correct location.

The last example merits extra clarification: while `DESTDIR` is a widely-used convention, it is by no means mandatory. Since Makefiles are fairly free-form and can call arbitrary programs, it's crucial to inspect your project's specific `Makefile` to discover the option that it exposes to control where artefacts will be placed when `make install` is called, and adjust the contents of the `override-build` script to reflect this. Failure to do so will frequently *not* result in a build error because the artefacts will be installed in a standard location like `/usr/local` *in the build system*, which is typically an LXD instance or a Multipass VM.

CHAPTER

THREE

REFERENCE

This section of the documentation contains an in-depth description of Rockcraft's components, commands and keywords.

3.1 rockcraft.yaml

3.1.1 Part properties

after

Type: array of unique strings with at least 1 item

Step: build

Specifies a list of parts that a given part will be built *after*.

build-environment

Type: build-environment-grammar

Step: build

The environment variables to be defined in the build environment specified as a list of key-value pairs.

Example:

```
build-environment:
  - MESSAGE: "Hello world"
  - NAME: "Craft Parts"
```

build-packages

Type: grammar-array

Step: build

The system packages to be installed in the build environment before the build is performed. These are installed using the host's native package manager, such as `apt` or `dnf`, and they provide libraries and executables that the part needs during the build process.

build-snaps

Type: grammar-array

Step: build

The snaps to be installed in the build environment before the build is performed. These provide libraries and executables that the part needs during the build process.

organize

Type: ordered dictionary mapping strings to strings

Step: stage

Describes how files in the building area should be represented in the staging area.

In the following example, the `hello.py` file in the build area is copied to the `bin` directory in the staging area and renamed to `hello`:

```
organize:
  hello.py: bin/hello
```

override-build

Type: string

Step: pull

A string containing commands to be run in a shell instead of performing those defined by the plugin for the build step.

override-prime

Type: string

Step: pull

A string containing commands to be run in a shell instead of performing the standard actions for the prime step.

override-pull

Type: string

Step: pull

A string containing commands to be run in a shell instead of performing the standard actions for the pull step.

override-stage

Type: string

Step: pull

A string containing commands to be run in a shell instead of performing the standard actions for the stage step.

parse-info

Type: string

Step: all

plugin

Type: string

Step: all steps

The plugin used to build the part. Available plugins include the following:

Name	Note
ant	Apache Ant
autotools	Autotools
cmake	CMake
dotnet	.Net
dump	Simple file unpacking
go	Go
make	Make
maven	Apache Maven
meson	Meson
nil	No default actions
npm	NPM
python	Python package
rust	Rust with Cargo
scons	SCons

prime

Type: array of unique strings with at least 1 item

Step: prime

The files to copy from the staging area to the priming area, see *Specifying paths*.

source

Type: grammar-string

Step: pull

The location of the source code and data.

source-branch

Type: string

Step: pull

The branch in the source repository to use when pulling the source code.

source-checksum

Type: string

Step: pull

For plugins that use files, this key contains a checksum value to be compared against the checksum of the downloaded file.

source-commit

Type: string

Step: pull

The commit to use to select a particular revision of the source code obtained from a repository.

source-depth

Type: integer

Step: pull

The number of commits in a repository's history that should be fetched instead of the complete history.

source-subdir

Type: string

Step: pull

The subdirectory in the unpacked sources where builds will occur.

Note: This key restricts the build to the subdirectory specified, preventing access to files in the parent directory and elsewhere in the file system directory structure.

---

source-submodules

Type: array of unique strings with 0 or more items

Step: pull

The submodules to fetch in the source repository.

source-tag

Type: string

Step: pull

The tag to use to select a particular revision of the source code obtained from a repository.

source-type

Type: one of “deb”, “file”, “git”, “local”, “rpm”, “snap”, “tar”, “zip”

Step: pull

The type of container for the source code. If not specified, Craft Parts will attempt to auto-detect the source type. A list of supported formats can be found in the `craft_parts.sources` file.

stage

Type: array of unique strings with at least 1 item

Step: stage

The files to copy from the building area to the staging area, see *Specifying paths*.

stage-packages

Type: grammar-array

Step: stage

The packages to install in the staging area for deployment with the build products. These provide libraries and executables to support the deployed part.

This keyword also support supports Chisel slices.

To install a package slice instead of the whole package, simply follow the Chisel convention *<packageName>_<sliceName>*.

NOTE: at the moment, it is not possible to mix packages and slices in the same stage-packages field.

stage-snaps

Type: grammar-array

Step: stage

The snaps to install in the staging area for deployment with the build products. These provide libraries and executables to support the deployed part.

Summary of keys and steps

The following table shows the keys that are used in each build step. The `plugin` and `parse-info` keys apply to all steps.

Pull	Build	Stage	Prime
source	after	stage	prime
source-checksum	build-attributes	stage-snaps	
source-branch	build-environment	stage-packages	
source-commit	build-packages		
source-depth	build-snaps		
source-submodules	organize		
source-subdir			
source-tag			
source-type			
override-pull	override-build	override-stage	override-prime

A Rockcraft project is defined in a YAML file named `rockcraft.yaml` at the root of the project tree in the filesystem.

This Reference section is for when you need to know which options can be used, and how, in this `rockcraft.yaml` file.

3.1.2 Format specification

`name`

Type: string

Required: Yes

The name of the rock. This value must conform with Pebble's format for layer files, meaning that the `name`:

- must start with a lowercase letter [a-z];
- must contain only lowercase letters [a-z], numbers [0-9] or hyphens;
- must not end with a hyphen, and must not contain two or more consecutive hyphens.

title

Type: string

Required: No

The human-readable title of the rock. If omitted, defaults to `name`.

summary

Type: string

Required: Yes

A short summary describing the rock.

description

Type: string

Required: Yes

A longer, possibly multi-line description of the rock.

version

Type: string

Required: Yes

The rock version, used to tag the OCI image and name the rock file.

base

Type: One of `ubuntu@20.04 | ubuntu@22.04 | ubuntu@24.04 | bare`

Required: Yes

The base system image that the rock's contents will be layered on. This is also the system that will be mounted and made available when using Overlays. The special value `bare` means that the rock will have no base system at all, which is typically used with static binaries or *Chisel slices*.

Note: The notation "ubuntu:<channel>" is also supported for some channels, but this format is deprecated and should be avoided.

Note: Base `ubuntu@24.04` is still unstable and under active development. To use it, `build-base` *must* be `devel`.

---

build-base

Type: One of `ubuntu@20.04 | ubuntu@22.04 | devel`

Required: Yes, if `base` is `bare`

The system and version that will be used during the rock's build, but not included in the final rock itself. It comprises the set of tools and libraries that Rockcraft will use when building the rock's contents. This field is mandatory if `base` is `bare`, but otherwise it is optional and defaults to the value of `base`.

Note: The notation "ubuntu:<channel>" is also supported for some channels, but this format is deprecated and should be avoided.

Note: `devel` is a "special" value that means "the next Ubuntu version, currently in development". This means that the contents of this system changes frequently and should not be relied on for production rocks.

---

license

Type: string, in SPDX format

Required: Yes

The license of the software packaged inside the rock. This must match the SPDX format, but is case insensitive (e.g. both `MIT` and `mit` are valid).

run-user

Type: string

Required: No

The default OCI user. It must be a supported shared user. Currently, the only supported shared user is "_daemon_" (with UID/GID 584792). It defaults to "root" (with UID 0).

environment

Type: dict

Required: No

A set of key-value pairs specifying the environment variables to be added to the base image's OCI environment.

Note: String interpolation is not yet supported so any attempts to dynamically define environment variables with `$` will end in a project validation error.

---

services

Type: dict, following the Pebble Layer Specification format

Required: No

A list of services for the Pebble entrypoint. It uses Pebble's layer specification syntax exactly, with each entry defining a Pebble service. For each service, the `override` and `command` fields are mandatory, but all others are optional.

entrypoint-service

Type: string

Required: No

The optional name of the Pebble service to serve as the OCI entrypoint. If set, this makes Rockcraft extend `["/bin/pebble", "enter", "--verbose"]` with `["--args", "<serviceName>"]`. The command of the Pebble service must contain an optional argument that will become the OCI CMD.

> **Warning:** This option must only be used in cases where the targeted deployment environment has unalterable assumptions about the container image's entrypoint.

checks

Type: dict, following the Pebble Layer Specification format

Required: No

A list of health checks that can be configured to restart Pebble services when they fail. It uses Pebble's layer specification syntax, with each entry corresponding to a check. Each check can be one of three types: `http`, `tcp` or `exec`.

platforms

Type: dict

Required: Yes

The set of architecture-specific rocks to be built. Supported architectures are: `amd64`, `arm64`, `armhf`, `i386`, `ppc64el`, `riscv64` and `s390x`.

Entries in the `platforms` dict can be free-form strings, or the name of a supported architecture (in Debian format).

> **Warning:** **All** target architectures must be compatible with the architecture of the host where Rockcraft is being executed (i.e. emulation is not supported at the moment).

platforms.<entry>.build-on

Type: list[string]

Required: Yes, if `build-for` is specified *or* if `<entry>` is not a supported architecture name.

Host architectures where the rock can be built. Defaults to `<entry>` if that is a valid, supported architecture name.

platforms.<entry>.build-for

Type: string | list[string]

Required: Yes, if `<entry>` is not a supported architecture name.

Target architecture the rock will be built for. Defaults to `<entry>` that is a valid, supported architecture name.

Note: At the moment Rockcraft will only build for a single architecture, so if provided `build-for` must be a single string or a list with exactly one element.

---

parts

Type: dict

Required: Yes

The set of parts that compose the rock's contents (see *Parts*).

Note: The fields `entrypoint`, `cmd` and `env` are not supported in Rockcraft. All rocks have Pebble as their entrypoint, and thus you must use `services` to define your container application.

---

extensions

Type: list[string]

Required: No

Extensions to enable when building the ROCK.

Currently supported extensions:

- `flask-framework`

3.1.3 Example

```
name: hello
title: Hello World
summary: An Hello World rock
description: |
  This is just an example of a Rockcraft project
  for a Hello World rock.
version: latest
base: bare
```

(continues on next page)

(continued from previous page)

```
build-base: ubuntu@22.04
license: Apache-2.0
run-user: _daemon_
environment:
  FOO: bar
services:
  hello:
    override: replace
    command: /usr/bin/hello -t
    environment:
      VAR1: value
      VAR2: "other value"
platforms:
  amd64:
  armhf:
    build-on: ["armhf", "arm64"]
  ibm:
    build-on: ["s390x"]
    build-for: s390x
parts:
  hello:
    plugin: nil
    stage-packages:
      - hello
```

3.2 Rockcraft commands

3.2.1 build

Build artefacts defined for a part. If part names are specified only those parts will be built, otherwise all parts will be built.

Usage

rockcraft build [options] <part-name>

Required

part-name
: Optional list of parts to process.

Options

--build-for
Set architecture to build for.

--debug
Shell into the environment if the build fails.

--destructive-mode
Build in the current host.

--platform
Set platform to build for.

--shell
Shell into the environment in lieu of the step to run.

--shell-after
Shell into the environment after the step has run.

--use-lxd
Build in a LXD container.

Global options

-h or --help
Show this help message and exit.

-q or --quiet
Only show warnings and errors, not progress.

-v or --verbose
Show debug information and be more verbose.

--verbosity
Set the verbosity level to 'quiet', 'brief', 'verbose', 'debug' or 'trace'.

-V or --version
Show the application version and exit.

3.2.2 clean

Clean up artefacts belonging to parts. If no parts are specified, remove the packing environment.

Usage

```
rockcraft clean [options] <part-name>
```

Required

part-name
Optional list of parts to process.

Options

--destructive-mode
Build in the current host.

--use-lxd
Build in a LXD container.

Global options

-h or --help
Show this help message and exit.

-q or --quiet
Only show warnings and errors, not progress.

-v or --verbose
Show debug information and be more verbose.

--verbosity
Set the verbosity level to 'quiet', 'brief', 'verbose', 'debug' or 'trace'.

-V or --version
Show the application version and exit.

3.2.3 expand-extensions

Extensions listed rockcraft.yaml will be expanded and shown as output.

Usage

```
rockcraft expand-extensions [options]
```

Global options

-h or --help
Show this help message and exit.

-q or --quiet
Only show warnings and errors, not progress.

-v or --verbose
Show debug information and be more verbose.

--verbosity
Set the verbosity level to 'quiet', 'brief', 'verbose', 'debug' or 'trace'.

-V or --version
Show the application version and exit.

3.2.4 extensions

List available extensions and their corresponding bases.

Usage

```
rockcraft extensions [options]
```

Global options

-h or --help
Show this help message and exit.

-q or --quiet
Only show warnings and errors, not progress.

-v or --verbose
Show debug information and be more verbose.

--verbosity
Set the verbosity level to 'quiet', 'brief', 'verbose', 'debug' or 'trace'.

-V or --version
Show the application version and exit.

3.2.5 init

Initialise a rockcraft project by creating a minimalist, yet functional, rockcraft.yaml file in the current directory.

Usage

```
rockcraft init [options]
```

Options

--name
The name of the rock; defaults to the directory name.

--profile
Use the specified project profile (defaults to 'simple').

Global options

-h or --help
Show this help message and exit.

-q or --quiet
Only show warnings and errors, not progress.

-v or --verbose
Show debug information and be more verbose.

--verbosity
Set the verbosity level to 'quiet', 'brief', 'verbose', 'debug' or 'trace'.

-V or --version
Show the application version and exit.

3.2.6 list-extensions

List available extensions and their corresponding bases.

Usage

```
rockcraft list-extensions [options]
```

Global options

-h or --help
Show this help message and exit.

-q or --quiet
Only show warnings and errors, not progress.

-v or --verbose
Show debug information and be more verbose.

--verbosity
Set the verbosity level to 'quiet', 'brief', 'verbose', 'debug' or 'trace'.

-V or --version
Show the application version and exit.

3.2.7 overlay

Execute operations defined for each part on a layer over the base filesystem, potentially modifying its contents.

Usage

```
rockcraft overlay [options] <part-name>
```

Required

part-name
Optional list of parts to process.

Options

--build-for
Set architecture to build for.

--debug
Shell into the environment if the build fails.

--destructive-mode
Build in the current host.

`--platform`
Set platform to build for.

`--shell`
Shell into the environment in lieu of the step to run.

`--shell-after`
Shell into the environment after the step has run.

`--use-lxd`
Build in a LXD container.

Global options

`-h` or `--help`
Show this help message and exit.

`-q` or `--quiet`
Only show warnings and errors, not progress.

`-v` or `--verbose`
Show debug information and be more verbose.

`--verbosity`
Set the verbosity level to 'quiet', 'brief', 'verbose', 'debug' or 'trace'.

`-V` or `--version`
Show the application version and exit.

3.2.8 pack

Process parts and create the final artefact.

Usage

```
rockcraft pack [options] <part-name>
```

Required

`part-name`
Optional list of parts to process.

Options

`--build-for`
Set architecture to build for.

`--debug`
Shell into the environment if the build fails.

`--destructive-mode`
Build in the current host.

`--output` or `-o`
Output directory for created packages.

--platform
Set platform to build for.

--use-lxd
Build in a LXD container.

Global options

-h or --help
Show this help message and exit.

-q or --quiet
Only show warnings and errors, not progress.

-v or --verbose
Show debug information and be more verbose.

--verbosity
Set the verbosity level to 'quiet', 'brief', 'verbose', 'debug' or 'trace'.

-V or --version
Show the application version and exit.

3.2.9 prime

Prepare the final payload to be packed, performing additional processing and adding metadata files. If part names are specified only those parts will be primed. The default is to prime all parts.

Usage

```
rockcraft prime [options] <part-name>
```

Required

part-name
Optional list of parts to process.

Options

--build-for
Set architecture to build for.

--debug
Shell into the environment if the build fails.

--destructive-mode
Build in the current host.

--platform
Set platform to build for.

--shell
Shell into the environment in lieu of the step to run.

--shell-after
Shell into the environment after the step has run.

--use-lxd
Build in a LXD container.

Global options

-h or --help
Show this help message and exit.

-q or --quiet
Only show warnings and errors, not progress.

-v or --verbose
Show debug information and be more verbose.

--verbosity
Set the verbosity level to 'quiet', 'brief', 'verbose', 'debug' or 'trace'.

-V or --version
Show the application version and exit.

3.2.10 pull

Download or retrieve artefacts defined for a part. If part names are specified only those parts will be pulled, otherwise all parts will be pulled.

Usage

```
rockcraft pull [options] <part-name>
```

Required

part-name
Optional list of parts to process.

Options

--build-for
Set architecture to build for.

--debug
Shell into the environment if the build fails.

--destructive-mode
Build in the current host.

--platform
Set platform to build for.

--shell
Shell into the environment in lieu of the step to run.

--shell-after
Shell into the environment after the step has run.

--use-lxd
Build in a LXD container.

Global options

-h or --help
Show this help message and exit.

-q or --quiet
Only show warnings and errors, not progress.

-v or --verbose
Show debug information and be more verbose.

--verbosity
Set the verbosity level to 'quiet', 'brief', 'verbose', 'debug' or 'trace'.

-V or --version
Show the application version and exit.

3.2.11 stage

Stage built artefacts into a common staging area. If part names are specified only those parts will be staged. The default is to stage all parts.

Usage

```
rockcraft stage [options] <part-name>
```

Required

part-name
Optional list of parts to process.

Options

--build-for
Set architecture to build for.

--debug
Shell into the environment if the build fails.

--destructive-mode
Build in the current host.

--platform
Set platform to build for.

--shell
Shell into the environment in lieu of the step to run.

--shell-after
Shell into the environment after the step has run.

--use-lxd
Build in a LXD container.

Global options

-h or --help
Show this help message and exit.

-q or --quiet
Only show warnings and errors, not progress.

-v or --verbose
Show debug information and be more verbose.

--verbosity
Set the verbosity level to 'quiet', 'brief', 'verbose', 'debug' or 'trace'.

-V or --version
Show the application version and exit.

3.2.12 version

Show the application version and exit

Usage

```
rockcraft version [options]
```

Global options

-h or --help
Show this help message and exit.

-q or --quiet
Only show warnings and errors, not progress.

-v or --verbose
Show debug information and be more verbose.

--verbosity
Set the verbosity level to 'quiet', 'brief', 'verbose', 'debug' or 'trace'.

-V or --version
Show the application version and exit.

3.2.13 Lifecycle commands

Lifecycle commands can take an optional parameter `<part-name>`. When a part name is provided, the command applies to the specific part. When no part name is provided, the command applies to all parts.

build
: Build artefacts defined for a part.

clean
: Remove a part's assets.

overlay
: Create part layers over the base filesystem.

pack
: Create the final artefact.

prime
: Prime artefacts defined for a part.

pull
: Download or retrieve artefacts defined for a part.

stage
: Stage built artefacts into a common staging area.

3.2.14 Extension commands

expand-extensions
: Expand extensions in snapcraft.yaml.

extensions
: List available extensions for all supported bases.

list-extensions
: List available extensions for all supported bases.

3.2.15 Other commands

init
: Initialise a rockcraft project.

version
: Show the application version and exit.

3.3 Extensions

Just as the Snapcraft extensions are designed to simplify Snap creation, Rockcraft extensions are crafted to expand and modify the user-provided rockcraft project file, aiming to minimise the boilerplate code when initiating a new rock.

3.3.1 The `flask-framework` extension

The Flask extension streamlines the process of building Flask application rocks.

It facilitates the installation of Flask application dependencies, including Gunicorn, in the rock image. Additionally, it transfers your project files to `/flask/app` within the rock image.

A statsd-exporter is installed alongside the Gunicorn server to export Gunicorn server metrics.

3.3.2 The `django-framework` extension

The Django extension is similar to the flask-framework extension but tailored for Django applications.

3.4 Rockcraft plugins

This section contains an in-depth description of the plugins available in Rockcraft.

3.4.1 Dump plugin

The Dump plugin can be used for any project where you want to include existing files from somewhere and keep the content as is. Its source can be a local directory, a remote repository, or a URL. Common use cases include:

- Include static files like scripts or media from a local directory.
- Download and unpack pre-compiled proprietary software from a remote URL.
- Use git to clone a remote SDK, dataset, or model.

Keywords

This plugin uses the common *plugin* keywords as well as those for *sources*.

You must specify at least the *source* keyword. The *source-type* keyword is optional, but recommended, as it is used to specify how the source should be handled.

Dependencies

This plugin has no dependencies.

How it works

During the build step, the plugin performs the following actions:

- Check the `source-type` keyword to determine the type of the source if specified, otherwise, it will try to guess the type based on the `source`.
- Download the file or clone the repository if the `source` is a remote location.
- Copy the file or directory if the `source` is a local location.
- Unpack the file if it is an archive specified by the `source-type` keyword.
- Copy all contents and preserve the directory structure to the part's install directory.

3.4.2 Maven plugin

The Maven plugin builds Java projects using the Maven build tool.

After a successful build, this plugin will:

- Create `bin/` and `jar/` directories in `$CRAFT_PART_INSTALL`.
- Find the `java` executable provided by the part and link it as `$CRAFT_PART_INSTALL/bin/java`.
- Hard link the `.jar` files generated in `$CRAFT_PART_SOURCE` to `$CRAFT_PART_INSTALL/jar`.

Keywords

In addition to the common *plugin* and *sources* keywords, this plugin provides the following plugin-specific keywords:

maven-parameters

Type: list of strings

Used to add additional parameters to the `mvn package` command line.

Environment variables

This plugin reads the `http_proxy` and `https_proxy` variables from the environment to configure Maven proxy access. A comma-separated list of hosts that should not be accessed via proxy is read from the `no_proxy` environment variable.

Please refer to Configuring Apache Maven for a list of environment variables used to configure Maven.

Dependencies

The plugin expects Maven to be available on the system as the `mvn` executable, unless a part named `maven-deps` is defined. In this case, the plugin will assume that this part will stage the `mvn` executable to be used in the build step.

Note that the Maven plugin does not make a Java runtime available in the target environment. This must be handled by the developer when defining the part, according to each application's runtime requirements.

3.4.3 Python plugin

The Python plugin can be used for Python projects where you would want to do any of the following things:

- Import Python modules with a `requirements.txt` file.
- Build a Python project that has a `setup.py` or `pyproject.toml` file.
- Install packages using **`pip`**.

Keywords

This plugin uses the common *plugin* keywords as well as those for *sources*.

Additionally, this plugin provides the plugin-specific keywords defined in the following sections.

python-requirements

Type: list of strings

List of paths to requirements files.

python-constraints

Type: list of strings

List of paths to constraint files.

python-packages

Type: list

A list of dependencies to install from PyPI. If needed, **`pip`**, **`setuptools`** and **`wheel`** can be upgraded here.

Environment variables

This plugin also sets environment variables in the build environment. These are defined in the following sections.

PARTS_PYTHON_INTERPRETER

Default value: python3

The interpreter binary to search for in `PATH`.

PARTS_PYTHON_VENV_ARGS

Default value: (empty string)

Additional arguments for venv.

Dependencies

Since none of the bases that are available for rocks contain a default Python installation, including a Python interpreter in Rockcraft projects is mandatory. The plugin also requires the `venv` module to create the virtual environment where Python packages are installed at build time.

The easiest way to do this is to include the `python3-venv` package in the `stage-packages` of the part that uses the Python plugin. This will pull in the default Python interpreter for the `build-base`, like Python 3.10 for Ubuntu 22.04. However, other versions can be used by explicitly declaring them - here's an example that uses `python3.12-venv` from the Deadsnakes ppa:

```
package-repositories:
  - type: apt
    ppa: deadsnakes/ppa
    priority: always

parts:
  my-part:
    plugin: python
    source: .
    stage-packages: [python3.12-venv]
```

How it works

During the build step, the plugin performs the following actions:

- It creates a virtual environment directly into the `${CRAFT_PART_INSTALL}` directory.
- It uses **`pip`** to install the required Python packages as configured in the `python-requirements`, `python-constraints` and `python-packages` keywords.
- If the source contains a `setup.py` or `pyproject.toml` file, those files are used to install the dependencies specified by the package itself.

3.4.4 Rust plugin

The Rust plugin can be used for Rust projects that use the Cargo build system.

Keywords

In addition to the common *plugin* and *sources* keywords, this plugin provides the following plugin-specific keywords:

rust-channel

Type: string **Default:** stable

Used to select which Rust channel or version to use. It can be one of "stable", "beta", "nightly" or a version number. If you want to use a specific nightly version, use this format: `"nightly-YYYY-MM-DD"`. If you don't want this plugin to install Rust toolchain for you, you can put `"none"` for this option.

rust-features

Type: list of strings

Features used to build optional dependencies. This is equivalent to the `--features` option in Cargo.

You can also use `["*"]` to select all the features available in the project.

Note: This option does not override any default features specified by the project itself.

If you want to override the default features, please see the *rust-no-default-features* option below.

---

rust-no-default-features

Type: boolean **Default:** false

If this option is set to `true`, the default features specified by the project will be ignored.

You can then use the *rust-features* keyword to specify any features you wish to override.

rust-path

Type: list of strings **Default:** .

The path to the package root (that contains the `Cargo.toml` file). This is equivalent to the `--manifest-path` option in Cargo.

rust-use-global-lto

Type: boolean **Default:** false

Whether to use global LTO. This option may significantly impact the build performance but reducing the final binary size and improve the runtime performance. This will forcibly enable LTO for all the crates you specified, regardless of whether the projects have the LTO option enabled in the Cargo.toml file.

This is equivalent to the `lto = "fat"` option in the Cargo.toml file.

If you want better runtime performance, see the *Performance tuning* section below.

rust-ignore-toolchain-file

Type: boolean **Default:** false

Whether to ignore the `rust-toolchain.toml` and `rust-toolchain` file. The upstream project can use this file to specify which Rust toolchain to use and which component to install. If you don't want to follow the upstream project's specifications, you can put true for this option to ignore the toolchain file.

rust-cargo-parameters

Type: list of strings **Default:** []

Append additional parameters to the Cargo command line.

rust-inherit-ldflags

Type: boolean **Default:** false

Whether to inherit the LDFLAGS from the environment. This option will add the LDFLAGS from the environment to the Rust linker directives.

Cargo build system and Rust compiler by default do not respect the `LDFLAGS` environment variable. This option will cause the craft-parts plugin to forcibly add the contents inside the `LDFLAGS` to the Rust linker directives by wrapping and appending the `LDFLAGS` value to `RUSTFLAGS`.

Note: You may use this option to tune the Rust binary in a classic Snap to respect the Snap linkage, so that the binary will not find the libraries in the host filesystem.

Here is an example on how you might do this on core22:

```
parts:
  my-classic-app:
    plugin: rust
    source: .
    rust-inherit-ldflags: true
    build-environment:
      - LDFLAGS: >
          -Wl,-rpath=\$ORIGIN/lib:/snap/core22/current/lib/$CRAFT_ARCH_TRIPLET_BUILD_FOR
          -Wl,-dynamic-linker=$(find /snap/core22/current/lib/$CRAFT_ARCH_TRIPLET_BUILD_
↪FOR -name 'ld*.so.*' -print | head -n1)
```

Environment variables

This plugin sets the PATH environment variable so the Rust compiler is accessible in the build environment.

Some environment variables may also influence the Rust compiler or Cargo build tool. For more information, see Cargo documentation for the details.

Dependencies

By default this plugin uses Rust toolchain binaries from the Rust upstream. If this is not desired, you can set `rust-deps: ["rustc", "cargo"]` and `rust-channel: "none"` in the part definition to override the default behaviour.

Performance tuning

Warning: Keep in mind that due to individual differences between different projects, some of the optimisations may not work as expected or even incur performance penalties. YMMV.

Some programs may even behave differently or crash if aggressive optimisations are used.

Many Rust programs boast their performance over similar programs implemented in other programming languages. To get even better performance, you might want to follow the tips below.

- Use the *rust-use-global-lto* option to enable LTO support. This is suitable for most projects. However, analysing the whole program during the build time requires more memory and CPU time.
- Specify `codegen-units=1` in `Cargo.toml` to reduce LLVM parallelism. This may sound counter-intuitive, but reducing code generator threads could improve the quality of generated machine code. This option will also reduce the build time performance since the code generator uses only one thread per translation unit.
- Disable `incremental=true` in `Cargo.toml` to improve inter-procedural optimisations. Many projects may have already done this for the release profile. You should check if that is the case for your project.
- (Advanced) Perform cross-language LTO. This requires installing the correct version of LLVM/Clang and setting the right environment variables. You must know which LLVM version of your selected Rust toolchain is using.

For example, Rust 1.71 uses LLVM 16 because you can see it bundles a `libLLVM-16-rust-1.71.1-stable.so` file in the `lib` directory. In this case, you want to install `clang-16` and `lld-16` from the Ubuntu archive.

- **You will need to set these environment variables for Clang:**

```
parts:
  my-app:
    plugin: rust
    source: .
    build-packages:
      - clang-16
      - lld-16
    build-environment:
      - CC: clang-16
      - CXX: clang++-16
      - CFLAGS: -flto=fat
      - CXXFLAGS: -flto=fat
      - RUSTFLAGS: "-Cembed-bitcode=yes -Clinker-plugin-lto -Clinker=clang-
↪16 -Clink-arg=-flto -Clink-arg=-fuse-ld=lld"
```

For some projects that manipulate the object files during the build, you may also need:

```
export NM=llvm-nm-16
export AR=llvm-ar-16
export RANLIB=llvm-ranlib-16
```

You can refer to the rustc documentation for more information on the meaning of those options.

- You will need significantly more memory and CPU time for large projects to build and link. For instance, Firefox under full LTO requires about 62 GiB of memory to pass the linking phase.

3.5 Parts and Steps

Parts and steps are the basic data types craft-parts will work with. Together, they define the lifecycle of a project (i.e. how to process each step of each part in order to obtain the final primed result).

3.5.1 Parts

When the `LifecycleManager` is invoked, parts are defined in a dictionary under the `parts` key. If the dictionary contains other keys, they will be ignored.

Permissions

Parts can declare read/write/execute permissions and ownership for the files they produce. This is achieved by adding a `permissions` subkey in the specific part:

```
# ...
parts:
  my-part:
    # ...
    permissions:
      - path: bin/my-binary
```

(continues on next page)

(continued from previous page)

```
      owner: 1111
      group: 2222
      mode: "755"
```

The `permissions` subkey is a list of permissions definitions, each with the following keys:

- `path`: a string describing the file(s) and dir(s) that this definition applies to. The path should be relative, and supports wildcards. This field is *optional* and its absence is equivalent to `"*"`, meaning that the definition applies to all files produced by the part;
- `owner`: an integer describing the numerical id of the owner of the files. This field is *optional* in the general case but *mandatory* if `group` is specified;
- `group`: an integer describing the numerical id of the group for the files. The semantics are otherwise the same as `owner`, including being *optional* in the general case and *mandatory* if `owner` is specified;
- `mode`: string describing the desired permissions for the files as a number in base 8. This field is *optional*.

3.5.2 Steps

Steps are used to establish plan targets and in informational data structures such as `StepInfo`. They are defined by the `Step` enumeration, containing entries for the lifecycle steps `PULL`, `OVERLAY`, `BUILD`, `STAGE`, and `PRIME`.

Step execution environment

Craft-parts defines the following environment for use during step processing and execution of user-defined scriptlets:

- `CRAFT_ARCH_TRIPLET`: The the machine-vendor-os platform triplet definition.
- `CRAFT_TARGET_ARCH`: The architecture we're building for.
- `CRAFT_PARALLEL_BUILD_COUNT`: The maximum number of concurrent build jobs to execute.
- `CRAFT_PROJECT_DIR`: The path to the current project's subtree in the filesystem.
- `CRAFT_PART_NAME`: The name of the part currently being processed.
- `CRAFT_PART_SRC`: The path to the part source directory. This is where sources are located after the `PULL` step.
- `CRAFT_PART_SRC_WORK`: The path to the part source subdirectory, if any. Defaults to the part source directory.
- `CRAFT_PART_BUILD`: The path to the part build directory. This is where parts are built during the `BUILD` step.
- `CRAFT_PART_BUILD_WORK`: The path to the part build subdirectory in case of out-of-tree builds. Defaults to the part source directory.
- `CRAFT_PART_INSTALL`: The path to the part install directory. This is where built artefacts are installed after the `BUILD` step.
- `CRAFT_OVERLAY`: The path to the part's layer directory during the `OVERLAY` step if overlays are enabled.
- `CRAFT_STAGE`: The path to the project's staging directory. This is where installed artefacts are migrated after the `STAGE` step.
- `CRAFT_PRIME`: The path to the final primed payload directory after the `PRIME` step.

Step output directories

Some of the environment variables above reference directories that are the output locations for specific steps. These are repeated below for fast reference:

- **PULL:**
  - `CRAFT_PART_SRC` locates the source of the part.
  - `CRAFT_PART_SRC_WORK` locates the source subdirectory if overridden.
- **OVERLAY:**
  - `CRAFT_OVERLAY` locates the combined overlay output from all parts.
- **BUILD:**
  - `CRAFT_PART_INSTALL` contains the location of the build output step. This directory is the expected location of `CARGO_INSTALL_ROOT` for Rust, `GOBIN` for go or `DESTDIR` for make.
- **STAGE:**
  - `CRAFT_STAGE` contains the expected location of all staged outputs.
- **PRIME:**
  - `CRAFT_PRIME` contains the path of the primed payload directory. This directory is shared by all parts.

CHAPTER

FOUR

EXPLANATION

This section of the documentation covers the concepts used by Rockcraft and the motivations behind its development.

4.1 Why use Rockcraft?

Getting past the technical matters surrounding Rockcraft, from a higher perspective, you might be asking *"but what is this after all?"* and *"why do I need it?"*.

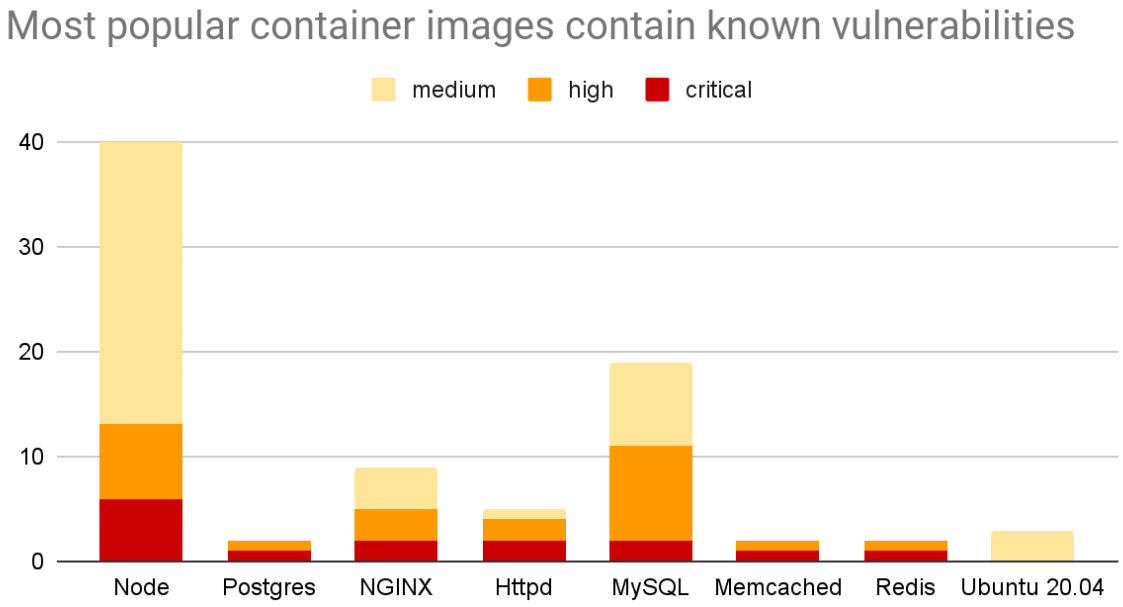
Let's then use this page to go a bit deeper into the concepts and definitions behind Rockcraft.

4.1.1 So why do I need Rockcraft?

Now, this is where things get interesting. To answer this question, we first need to look at the current state of the art with respect to the existing container image offerings out there.

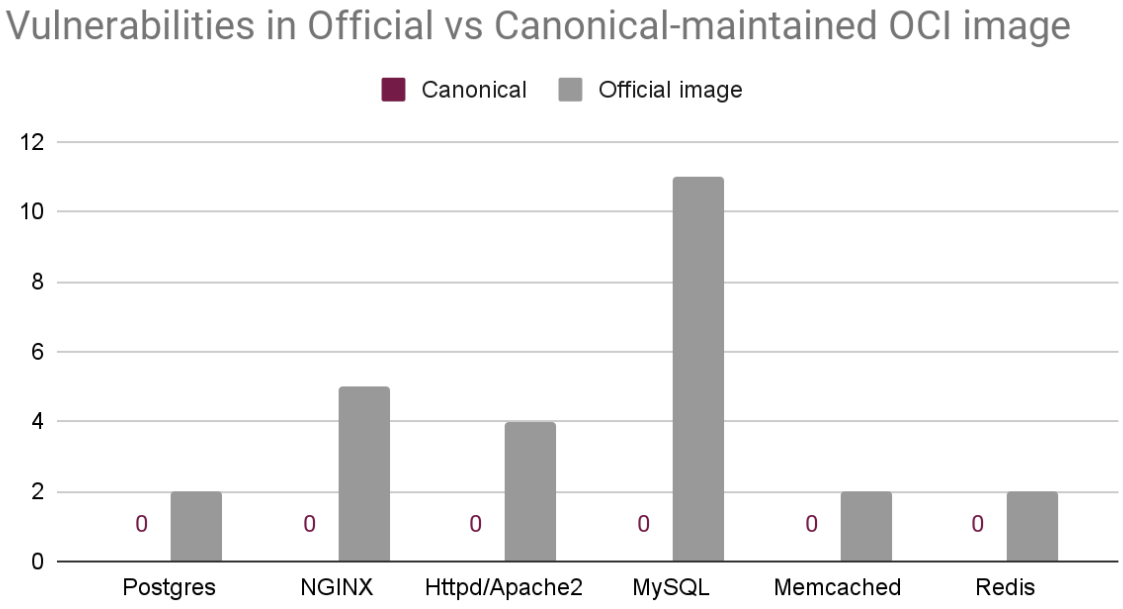
It is easy to find public studies (like Unit 42 / Znet and Snyk's state of open source security report 2020) where the findings state a concerning number of containers at risk deployed in cloud infrastructures.

In fact, both these studies and our own assessments (dated from December 2021) show that the most popular images in Docker Hub contain known vulnerabilities, with Ubuntu being the only one without any critical or high ones.



Updated December 2021 - Scan results with Snyk

Sure, consumers could venture to fix these vulnerabilities themselves, but not only would this increase the cost and proliferation of images, but it wouldn't be easy to accomplish due to the lack of expertise in the subject matter. The right approach is to actually fix the vulnerabilities at their source! And Canonical has already started doing this. If we compare some of the Docker Official container images vs some of the ones maintained by Canonical, we can verify that the latter have no high/critical vulnerabilities in them!



High and critical known vulnerabilities scanned with Snyk (December 2021)

So this is where the motivation for a new generation of OCI images (aka rocks) starts - the need for more secure

container images! And while this need might carry the biggest weight in the container users' demands, other values come into play when selecting the best container image, such as:

- stability
- size
- compliance
- provenance

You can find these values and their relevance in this report.

This brings us to the problem statement behind rocks:

> *How might we redesign secure container images for Kubernetes developers and application maintainers, considering the Top 10 Docker images are full of vulnerabilities, except Ubuntu?*

A rock is:

- **secure** and **stable**: based on the latest and greatest Ubuntu releases;
- **OCI-compliant**: compatible with all the popular container management tools (Docker, Kubernetes, etc.);
- **dependable**: built on top of Ubuntu, with a predictable release cadence and timely security updates;
- **production-grade**: tested and secured by default.

Do I need to use Rockcraft?

If you want to build a proper rock, yes, we'd recommend you do. This is not to say you wouldn't be able to build rock-like container images with your own tools, but Rockcraft has been developed precisely to offer an easy way to build production-grade container images.

Furthermore, Rockcraft is built on top of existing concepts and within the same family as Snapcraft and Charmcraft, such that its adoption becomes seamless for those already used to building Snaps and Charms.

4.2 Chisel

Chisel is a software tool for extracting well-defined portions (aka slices) of Debian packages into a filesystem.

Using the analogy of a tool to carve and cut stone, Chisel is used in Rockcraft to sculpt minimal collections of files that only include what is needed for the rock to function properly.

See *Cut existing slices with Chisel* for information about using the tool.

4.2.1 Package slices

Since Debian packages are simply archives that can be inspected, navigated and deconstructed, it is possible to define slices of packages that contain minimal, complementary, loosely-coupled sets of files based on package metadata and content. Such **package slices** are subsets of Debian packages, with their own content and set of dependencies to other internal and external slices.

The use of package slices provides Rockcraft with the ability to build minimal container images from the wider set of Ubuntu packages.

This image illustrates the simple case where, at a package level, package *B* depends on package *A*. However, there might be files in *A* that *B* doesn't actually need, but which are provided for convenience or completeness. By identifying the

files in *A* that are actually needed by *B*, we can divide *A* into slices that serve this purpose. In this example, the files in the package slice, *A_slice3*, are not needed for *B* to function. To make package *B* usable in the same way, it can also be divided into slices.

With these slice definitions in place, Chisel is able to extract a highly-customised and specialised slice of the Ubuntu distribution, which one could see as a block of stone from which we can carve and extract only the small and relevant parts that we need to run our applications, thus keeping rocks small and less exposed to vulnerabilities.

Defining slices

A package's slices can be defined via a YAML slice definitions file. Check the slice definitions reference for more information about this file's format.

Note: To find examples of existing slice definitions files, check the Chisel releases repository at https://github.com/canonical/chisel-releases. Contributions are welcome and encouraged.

4.3 Overlay step

The component parts of a rock are built in a sequence of five separate steps: pull, overlay, build, stage and prime.

The overlay step is specific to rocks and is configured with overlay parameters. To learn more about pull, build, stage and prime see *Part properties*

The overlay step provides the means to modify the base filesystem before the build step is applied. If `overlay-packages` is used, those packages will be installed first. `overlay-script` will run the provided script in this step. The location of the overlay is made available in the ${CRAFT_OVERLAY} environment variable. `overlay` can be used to specify which files will be migrated to the next steps, and when omitted its default value will be `"*"`.

4.3.1 Overlay Parameters

A part has three parameters that can be used to adjust how the overlay step works: `overlay-packages`, `overlay-script` and `overlay-filter`. `overlay-packages` and `overlay` (the filter parameter) behave much the same way as the related parameters on the `STAGE` step. `overlay-script` likewise behaves similarly to `override-stage`, including having access to the `craftctl` command.

An example parts section with overlay parameters looks as follows:

```
parts:
  part_with_overlay:
    plugin: nil
    overlay-packages:
      - ed
    overlay-script: |
      rm -f ${CRAFT_OVERLAY}/usr/bin/vi ${CRAFT_OVERLAY}/usr/bin/vim*
      rm -f ${CRAFT_OVERLAY}/usr/bin/emacs*
      rm -f ${CRAFT_OVERLAY}/bin/nano
    overlay:
      - bin
      - usr/bin
```

After running this part, the overlay layer (and the final package) will only contain ed as an editor, with vi/vim, emacs, and nano all having been removed.

4.4 Rocks

Rocks are **Ubuntu LTS-based container images** that are designed to meet cloud-native software's security, stability, and reliability requirements.

Rocks comply with the Open Container Initiative's (OCI) image format specification, and can be stored in any OCI-compliant container registry (e.g. DockerHub, ECR, ACR,..) and used by any OCI-compliant tool (e.g. Docker, Podman, Kubernetes,...).

Interoperability between rocks and other containers also extends to how container images are built, allowing rocks to be used as bases for existing build recipes, such as Dockerfiles, for further customisation and development.

4.4.1 What sets rocks apart?

- **Opinionated and consistent design**: all rocks follow the same design, aiming to minimise your full-stack disparity and adoption overhead, e.g.
  - *Pebble* **is the official entrypoint for all rocks**, providing a predictable and powerful abstraction layer between the user and the container application;
  - Rocks extend the OCI image information by including additional **metadata** inside each rock (e.g. at `/.rock/metadata.yaml`), allowing container applications to easily inspect the properties of the image they are running on, at execution time;
- **User-centric experience**: rocks are described in a *declarative format* and **built on top of familiar and reliable Ubuntu LTS images**, offering an open and up-to-date user experience;
- **Seamless chiselling experience**: rocks can be effortlessly *chiselled* using off-the-shelf primitives, harnessing all the advantages of "distroless" to deliver **compact and secure Ubuntu-based container images**.

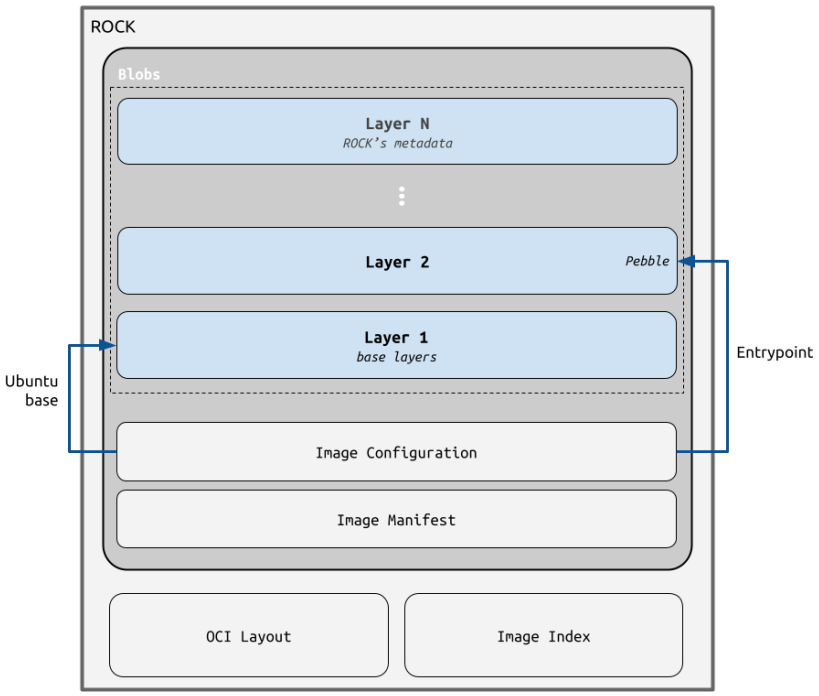
4.4.2 Design

Given their compliance with the OCI image specification, all rocks are constituted by OCI metadata (like the image's index, manifest and configuration) plus the actual OCI layers with the container filesystem contents.

Typically, container users won't be directly building or accessing the raw OCI components that form an image. However, these are frequently used as the underlying source of truth when inspecting container images with tools like Docker or skopeo. As an example, the command `docker inspect` will, in general, source the requested information from the image's OCI configuration.

On the other hand, the OCI layers are the literal filesystem contents that result from the user's instructions at image build time, and that can be accessed by the container application at runtime.

The following diagram depicts the different OCI components in the context of a rock, highlighting where the aforementioned design features (like the Pebble entrypoint) fit in.



4.5 Pebble

Important: Pebble is the default entrypoint for all rocks!

Similar to other well-known process managers such as *supervisord*, *runit*, or *s6*, Pebble is a service manager that enables the seamless orchestration of a collection of local service processes as an organised set. The main difference is that Pebble has been designed with custom-tailored features that significantly enhance the overall container experience, making it the ideal candidate for the container's init process (also known as the entrypoint, with PID=1).

4.5.1 Multiple processes in a container?

Containers' best practices advocate the separation of concerns and the adoption of a single service per container. With the introduction of Pebble as the rocks' entrypoint, this principle is elevated to new heights:

> *if multiple processes rely on shared dependencies and are tightly coupled together (i.e. they serve a single purpose and cannot be executed independently), then the best practice entails orchestrating them within the same container, with Pebble as their manager.*

This new notion addresses existing pain points arising from the excessive separation of concerns, which results in numerous container images whose entrypoints lack the ability to gracefully handle the underlying child processes. This is one of the main reasons behind the gradual shift in the best practices, as there is an increasing emphasis on adopting init processes such as tini, s6-overlay, or Pebble.

4.5.2 What to expect?

Pebble distinguishes itself from other similar tools (like tini and s6-overlay) by offering the following core features:

- **client-server model behind a single binary**: Pebble is injected into rocks as a single binary which acts both as a daemon and a client to itself;
- **declarative service definition**: the Pebble service processes (or simply *Pebble services*) are declaratively defined in YAML files called layers. Compared to imperative wrapper scripts (as suggested in the Docker documentation), this provides a much cleaner and less error-prone way to define the processes that should run inside the container.
- **services as first-class citizens**: unlike wrapper scripts, Pebble treats services as manageable units with a defined lifecycle and service-specific definitions for health monitoring, inter-service dependencies, restart policies, and much more;
- **layering**: Pebble can stack multiple layers (represented as YAML files) into a single Pebble plan where all services are defined. With this layering mechanism, existing services can be overridden or re-configured;
- **container-optimised init process**: as a rock's PID 1, Pebble acts as an init process and thus offers:
  - support for multiple child processes,
  - reaping and subreaping,
  - signal forwarding,
  - graceful shutdown,
  - log rotation,
  - run the Pebble daemon and client commands in a single operation;
- **consistent user experience**: since every rock has Pebble as its entrypoint, a predictable and consistent user experience is guaranteed;
- **embedded utilities**: regardless of the rock's contents, Pebble offers a comprehensive suite of commands for inspecting and interacting with the container. These commands are especially useful for *Chiselled Rocks*, as they encompass functionalities such as listing and deleting files, creating directories, and inspecting Pebble services, among others.

4.5.3 Creating services

Rockcraft follows the Pebble layer specification to the letter, with Pebble services defined in *rockcraft.yaml*. *How to convert an entrypoint to a Pebble layer* provides an example of how to convert a Docker entrypoint to a Pebble layer.

4.6 From prime step to OCI layer

Rockcraft is a tool that creates OCI images using the same concepts and mechanisms that create snaps and charms: the lifecycle language from Craft Parts. There is a significant difference between the way the Craft lifecycle works and the OCI specification, and one of Rockcraft's jobs is to bridge the gap between these two worlds. This page describes how this is accomplished.

Note: It is not necessary to know these details to use the tool effectively, but they might illuminate some concepts and help understand *why* the contents of a given rock are the way they are.

Consider the following snippet of a `rockcraft.yaml` that creates a rock containing a bare-bones Python 3.10 interpreter:

```
# (...)
base: ubuntu@22.04

parts:
  python-part:
    plugin: nil
    stage-packages:
      - python3-minimal
```

This rock has Ubuntu 22.04 as its base and includes `python3-minimal`. Conceptually, this means that at build time Craft Parts will pull in the `python3-minimal` Ubuntu package and whatever dependencies it needs to work. Indeed, if we run `rockcraft prime --shell-after`, we can see the final contents ready to be packed in the prime directory - this is the directory available at build-time through the `${CRAFT_PRIME}` environment variable:

```
$ rockcraft prime --shell-after
$ cd ../prime
$ ls
bin  etc  lib  lib64  sbin  usr  var
$ ls usr/bin/
debconf                debconf-copydb          debconf-show  dpkg-divert             ␣
↪dpkg-realpath      dpkg-trigger  py3clean     python3
debconf-apt-progress  debconf-escape          dpkg          dpkg-maintscript-helper ␣
↪dpkg-split         perl          py3compile   python3.10
debconf-communicate   debconf-set-selections  dpkg-deb      dpkg-query              ␣
↪dpkg-statoverride  perl5.34.0    py3versions  update-alternatives
```

As we can see, the prime directory has the contents of the `python3-minimal` package but also many of its dependencies, direct and otherwise. Once the lifecycle is finished, Rockcraft packs the contents of the prime directory as a new OCI layer, directly as if the prime directory were the filesystem root `/`.

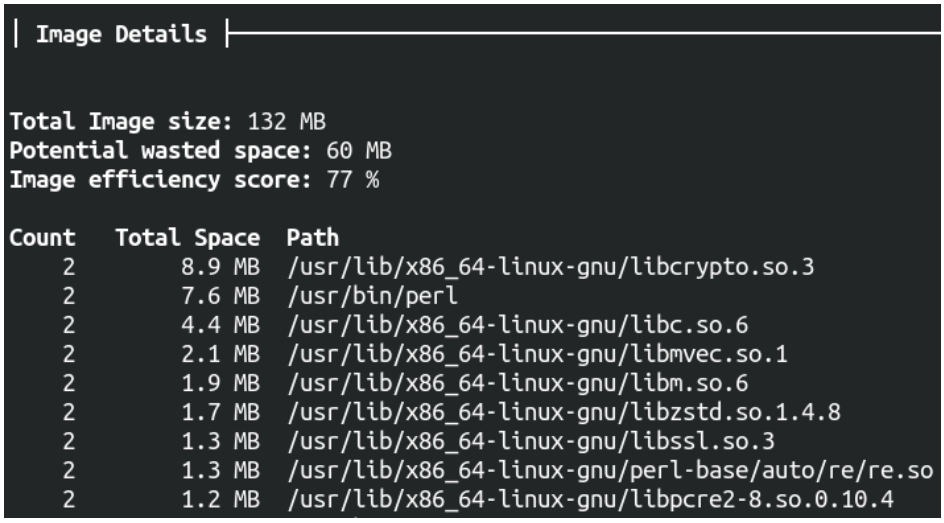
Note: The following sections only apply to rocks with Ubuntu bases - `bare` rocks don't need prime pruning nor `usrmerge` handling.

4.6.1 Pruning the `prime` directory

One consequence of the inclusion of a `stage-package`'s dependencies is that the prime directory ends up having many files that the base Ubuntu layer already has. This can be seen, for example, by using a tool like Dive:

What `dive` tells us is that about `60 MB` worth of files are *duplicated* between the base Ubuntu 22.04 layer and the "primed" layer: for example, the file `/usr/lib/x86_64-linux-gnu/libcrypto.so.3` exists both in the base layer (as part of the base Ubuntu system) and in the primed layer (pulled in by belonging to a package that is an indirect dependency of `python3-minimal`).

Starting from version `1.1.0`, Rockcraft "prunes" those files in the prime directory that also exist, with the same contents, ownership and permissions, in the base layer. The end result is semantically the same, because the layers are "stacked" together when creating containers from the rock. This "pruning" can be seen in the logs generated by Rockcraft:




```
(...)
Pruning: /root/prime/usr/lib/x86_64-linux-gnu/perl-base/unicore/lib/Sc/Gran.pl as it␣
↪exists on the base
Pruning: /root/prime/usr/lib/x86_64-linux-gnu/perl-base/unicore/lib/Bc/EN.pl as it␣
↪exists on the base
Pruning: /root/prime/usr/lib/x86_64-linux-gnu/perl-base/unicore/lib/PatSyn/Y.pl as it␣
↪exists on the base
Pruning: /root/prime/usr/lib/x86_64-linux-gnu/perl-base/unicore/lib/Dt/Init.pl as it␣
↪exists on the base
Pruning: /root/prime/usr/share/perl5/Debconf/Element/Noninteractive/Multiselect.pm as it␣
↪exists on the base
(...)
```

4.6.2 `usrmerge` and the lifecycle layer

After pruning, the contents of the prime directory are packed as a new OCI layer. In concrete terms, this means that the files and directories are added to a tar archive, which means that each file (or directory) gets added to the archive together with the "destination" path that it should have when the archive is extracted.

In most cases, the file's original path (relative to the root of the archive) and its destination path once extracted are the same, so the file that exists in the prime directory as `a/b/c/file.txt` should be extracted as `a/b/c/file.txt`.

However, there are cases where this "destination" path should be changed. For example, consider again the contents of the previous rock's prime directory:

```
$ ls -l
total 5
drwxr-xr-x 2 root root  3 Dec  7 20:30 bin
drwxr-xr-x 9 root root 10 Dec  7 20:30 etc
drwxr-xr-x 4 root root  4 Dec  7 20:30 lib
drwxr-xr-x 2 root root  2 Dec  7 20:30 lib64
drwxr-xr-x 2 root root  2 Dec  7 20:30 sbin
drwxr-xr-x 7 root root  7 Dec  7 20:30 usr
drwxr-xr-x 4 root root  4 Dec  7 20:30 var
```

(continues on next page)

(continued from previous page)

```
$ ls bin/
pebble
```

So `bin/` is a regular directory and contains the `pebble` binary, to serve as the rock's entrypoint. However, consider the base directory structure of an Ubuntu system:

```
$ ls -l /
total 84
lrwxrwxrwx   1 root root     7 ago 27  2022 bin -> usr/bin
drwxr-xr-x   5 root root  4096 nov 27 13:59 boot
drwxrwxr-x   2 root root  4096 ago 27  2022 cdrom
drwxr-xr-x  20 root root  5900 dez  7 19:57 dev
drwxr-xr-x 148 root root 12288 dez  7 15:15 etc
drwxr-xr-x   3 root root  4096 ago 27  2022 home
lrwxrwxrwx   1 root root     7 ago 27  2022 lib -> usr/lib
lrwxrwxrwx   1 root root     9 ago 27  2022 lib32 -> usr/lib32
lrwxrwxrwx   1 root root     9 ago 27  2022 lib64 -> usr/lib64
lrwxrwxrwx   1 root root    10 ago 27  2022 libx32 -> usr/libx32
```

`bin` is actually a symbolic link to `usr/bin`. This is the usrmerge, and it's been present in Ubuntu for many years now. Note that many other entries are also symlinks, like `lib` (to `usr/lib`) and `lib64` (to `usr/lib64`).

These two filesystems interact in a surprising way when stacked as OCI layers. If `bin/pebble` is added to the layer's archive as `bin/pebble` plus an entry for the `bin/` directory (which is a regular directory in the prime contents), once the two layers are stacked together in a container the `bin/` directory from the "prime layer" will *overwrite* the `bin -> usr/bin` symlink from the "base layer", which will make everything that assumed that the base binaries from `usr/bin/` would always be accessible through `bin/` break.

This issue is made much worse if the instead of breaking `bin/` we break the `lib*/` symlinks. Consider:

```
$ ldd /bin/bash
linux-vdso.so.1 (0x00007ffdf2af4000)
libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007f6053cbd000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f6053a00000)
/lib64/ld-linux-x86-64.so.2 (0x00007f6053e6b000)
```

The `bash` binary links to multiple dynamic libraries, but has a hardcoded path to the `/lib64/ld-linux-x86-64.so.2` dynamic loader. This loader is the program that does the actual finding of dynamic dependencies at runtime, and in an Ubuntu system its actual location is at `/usr/lib64/ld-linux-x86-64.so.2`. So if the `/lib64 -> usr/lib64` symlink is broken because the prime directory contains `lib64` as a regular directory, then the vast majority of the binaries in the final rock's base system will simply fail to run because their loader is no longer available at `/lib64/ld-linux-x86-64.so.2`.

To fix this, Rockcraft will take the base system into account when creating the archive for the prime layer. For instance, when considering `bin/pebble`, Rockcraft will:

1. Skip adding `bin/` as a regular directory, to avoid breaking the base system, and
2. Add `bin/pebble` as `usr/bin/pebble` in the layer archive.

This can be seen in the logs:

```
(...)
Creating new layer
(...)
Skipping /root/prime/bin because it exists as a symlink on the lower layer
```

(continues on next page)

(continued from previous page)

```
(...)
Adding to layer: /root/prime/bin/pebble as 'usr/bin/pebble'
(...)
```

Finally, as mentioned in the beginning none of this applies for rocks with `bare` bases, as there is no base system to contain duplicates that need to be pruned or symbolic links that need to be taken into account.

4.7 Filesets

Filesets are named collections of files and directories that can be migrated between steps in the process of *building a part*. They are used within Craft Parts to collect and filter the files and directories a part needs in its *stage* and *prime* processing *steps*.

Tools that depend on Craft Parts can use filesets to simplify and automate migration of files and directories between build steps. Users of those tools may need to know about filesets if they need to adjust the contents of the packages that the tools produce.

4.7.1 Fileset names

Internally, Craft Parts uses the `overlay`, `stage` and `prime` filesets to migrate files from all parts into the corresponding steps. For example, the *stage* fileset refers to all the files and directories that will be moved into the staging area ready for the *stage* step to be run.

4.7.2 Defining filesets

Filesets are defined using the `craft_parts.executor.Fileset` class which is used to perform operations on lists of file paths. This accepts a string containing the name of the fileset and a list of strings containing the file paths.

Filesets are defined for individual parts. The scope of each fileset is the part it is defined in. Filesets defined in one part cannot be used by another part, and filesets cannot be shared between parts.

Specifying paths

The paths used in filesets specify locations *relative* to the working directory where they will be used. Absolute paths cannot be used.

Paths can specify single files or directories, such as these examples:

- `usr/bin/hello`
- `usr/share`

They can also contain wildcards to select multiple files and directories, such as these:

- `usr/bin/*`
- `usr/lib/**/*.so*`

The second of these examples selects all the shared libraries in all nested directories inside the `usr/lib` directory.

Filesets can also *exclude* files and directories. This is done by prefixing a path with the `-` character, as in these examples:

- `-usr/bin/hello`
- `-usr/share/**/*.gz`

The second example selects and discards gzipped files in all nested directories inside the `usr/share` directory.

4.7.3 Using filesets

Built-in filesets for the *stage* and *prime* steps are both applied to the directory containing the artefacts from the *build* step. These are used to specify the files and directories to migrate to the *stage* and *prime* steps.

The contents of the filesets for these steps are specified using the *stage* and *prime* properties when defining a part.

The order in which paths are defined in a fileset is not important. The paths are collected so that all files and directories to be included are first located, then paths that exclude files and directories are used to filter out those that are not needed.

4.7.4 Summary

When defined:

- Filesets specify named collections of files and directories using file paths that can contain wildcards. Only relative paths are allowed.
- They can both include and exclude sets of files and directories.
- They are defined for a given part, not for multiple parts.

When used:

- Filesets are used at the start of a step to collect and filter artefacts from an earlier step.
- Their file paths are applied to the directory containing the artefacts from the earlier step.
- All files and directories included by filesets are first located, then filtered by the filesets that exclude paths.

4.8 Parts

In the Craft Parts framework, parts are descriptions of the components to be built and prepared for deployment in a payload, either individually or as part of a larger project containing many components.

When the Craft Parts framework is used to process a part on behalf of a tool or library, it performs some or all of the steps described in the *parts lifecycle*:

1. The *pull* step pulls the source code and dependencies from locations defined in the part and places them into a package cache.
2. The *overlay* step unpacks them into a base file system chosen from a collection of standard file system images.
3. The *build* step runs a suitable build tool for the sources to compile a set of build products or artefacts.
4. The *stage* step copies the build products for the part into a common area for all parts in a project.
5. The *prime* step copies the files to be deployed into an area for further processing.

Not all of these steps may be needed for every use case, and tools that use the Craft Parts framework can skip those that are not appropriate for their purposes.

Tools like Snapcraft and Charmcraft that use the concepts of parts to describe a build process typically accept specifications of parts in YAML format. This allows each part to be described in a convenient, mostly-declarative format. Libraries that use parts may use the underlying data structures to describe them.

4.8.1 Describing a part

Each part contains all the required information about a specific component, and is organised like a dictionary. Each piece of information is accessed by name using a property.

Generally, each part includes information about the following:

- Its *source* (where it is obtained from)
- Its *build dependencies* (snaps and packages)
- The *build process*
- How *build artefacts* are handled

Each of these are described in the following sections.

Source

The source for a part is described using the *source* property. This specifies a location where the source code or other information is to be *pulled* from. This may be a repository on a remote server, a directory on the build host, or some other location.

Additional properties are used to fine-tune the specification so that a precise description of the source location can be given, and also to specify the type of source to be processed.

Where the type of the source information cannot be automatically determined, the *source-type* property is used to explicitly specify the source format. This influences the way in which the source code or data is processed. A list of supported formats can be found in the `craft_parts.sources` file. These include repository types, such as `git`, archive formats such as `zip`, and `local` for files in the local file system.

If the source type represents a file, the *source-checksum* property can be used to provide a checksum value to be compared against the checksum of the downloaded file.

Parts with source types that describe repositories can also use additional properties to accurately specify where source code is found. The *source-branch*, *source-commit* and *source-tag* properties allow sources to be obtained from a specific branch, commit or tag.

Since some repositories can contain large amounts of data, the *source-depth* property can be used to specify the number of commits in a repository's history that should be fetched instead of the complete history. For repositories that use submodules, the *source-submodules* property can be used to fetch only those submodules that are needed.

The *source-subdir* property specifies the subdirectory in the unpacked sources where builds will occur. **Note:** This property restricts the build to the subdirectory specified, preventing access to files in the parent directory and elsewhere in the file system directory structure.

Build dependencies

The dependencies of a part are described using the *build-snaps* and *build-packages* properties. These specify lists of snaps and system packages to be installed before the part is built. If a part depends on other parts, the *after* property is used to specify these – see *Defining the build order*.

Snaps are referred to by the names that identify them in the Snap Store and can also include the channel information so that specific versions of snaps are used. For example, the `juju` snap could be specified as `juju/stable`, `juju/2.9/stable` or `juju/latest/stable` to select different versions.

System packages are referred to by the names that identify them on the host system, and they are installed using the host's native package manager, such as **`apt`** or **`dnf`**.

For example, a part that is built against the SDL 2 libraries could include the `libsdl2-dev` package in the *build-packages* property.

Build process

Each part specifies the name of a *plugin* using the `plugin` property to describe how it should be built. The available plugins are provided by the modules in the `craft_parts.plugins` package.

Plugins simplify the process of building source code written in a variety of programming languages using appropriate build systems, libraries and frameworks. If a plugin is not available for a particular combination of these attributes, a basic plugin can be used to manually specify the build actions to be taken, using the *override-build* property. This property can also be used to replace or extend the build process provided by a plugin.

When a plugin is used, it exposes additional properties that can be used to define behaviour that is specific to the type of project that the plugin supports. For example, the `cmake plugin` provides the `cmake-parameters` and `cmake-generator` properties that can be used to configure how **`cmake`** is used in the build process.

The *build-environment* property defines assignments to shell environment variables in the build environment. This is useful in situations where the build process of a part needs to be fine-tuned and can be configured by setting environment variables.

The result of the *build* step is a set of build artefacts or products that are the same as those that would be produced by manually compiling or building the software.

Build artefacts

At the end of the *build* step, the build artefacts can be organised before the *stage* step is run.

The *organize* property is used to customise how files are copied from the building area to the staging area. It defines an ordered dictionary that maps paths in the building area to paths in the staging area.

After the *build* step, the *stage* step is run to collect the artefacts from the build into a common staging area for all parts. Additional snaps and system packages that need to be deployed with the part are specified using the *stage-snaps* and *stage-packages* properties. Files to be deployed are specified using the *stage* property.

In the final *prime* step, the files needed for deployment are copied from the staging area to the priming area. During this step the `prime` property is typically used to exclude files in the staging area that are not required at run-time. This is especially useful for multi-part projects that include their own compilers or development tools.

Defining the build order

If a part depends on other parts in a project as build dependencies then it can use the *after* property to define this relationship. This property specifies a list containing the names of parts that it will be built after. The parts in the list will be *built and staged* before the part is built.

This is covered in detail in *Part processing order*.

4.8.2 How parts are built

As described in *Lifecycle details*, parts are built in a sequence of steps: *pull*, *overlay*, *build*, *stage* and *prime*.

A part is built in a clean environment to ensure that only the base file system and its dependencies are present, avoiding contamination from partial builds and side effects from other builds. The environment is a file system in a container where the root user's home directory is populated with a number of subdirectories and configured to use snaps.

Initially, before the *pull* step is run, the *working* directory contains a `project` directory containing the files for the project to be built.

The pull step

When the *pull* step is run the *sources* are obtained using the source definitions for each part. After the step, the *working* directory contains a `state` file to manage the state of the build and a number of subdirectories:

- `parts` is where individual parts for the project are prepared for build. The directory for each part in the `parts` directory contains `src`, `build` and `install` directories that will be used during the *build* step.
- `prime` will contain the finished build product later in the process.
- `project` contains the original, unmodified project files.
- `stage` will contain staged files after a build, before they are primed.

The standard actions for the *pull* step can be overridden or extended by using the *override-pull* key to describe a series of actions.

The build step

When the *build* step is run, each part in the `parts` subdirectory is processed in the order described in the *build order*. The plugin for the part will use the appropriate build system to build the part in its `build` subdirectory, using a copy of the files in its `src` subdirectory, and install the result in the part's `install` subdirectory. The files in the `install` directory will be organised according to the rules in the part's *organize* property.

After the *build* step is run, the directory for each part in the `parts` directory will contain updated `build` and `install` directories. The `build` directory will contain the build products, and the `install` directory will contain the files to be included in the payload.

Parts that depend on other parts will be built *after* their dependencies have been built and staged.

The stage step

When the *stage* step is run for a part, the contents of its `install` directory are copied into the common `stage` directory. Additionally, dependencies specified by the *stage-packages* and *stage-snaps* properties of the part are also unpacked into the `stage` directory.

The result is that `stage` directory can contain the files needed for the final payload as well as resources for other parts. If other parts need a part, such as a compiler, to be built and staged before they can be built, their *build* steps will run after the *stage* step for the part they depend on.

The prime step

When the *prime* step is run for a part, the contents of the common `stage` directory are filtered using the rules in the *prime* property and copied into the `prime` directory.

In a multi-part project the `stage` directory may contain resources that were required to build certain parts, or the build products may include files that are not needed at run-time. Using a separate `prime` directory in a separate *prime* step makes it possible to apply a filter to the build products.

4.9 Lifecycle details

Each part is built in *five separate steps*, each with its own input and output locations:

1. `PULL` — The source and external dependencies (such as package dependencies) for the part are retrieved from their stated location and placed into a package cache area.
2. `OVERLAY` — Any overlay packages are installed in an overlay of the filesystem base, and the overlay script is run. Finally, any overlay filters are applied.
3. `BUILD` — The part is built according to the particular part plugin and build override.
4. `STAGE` — The specified outputs from the `BUILD` step are copied into a unified staging area for all parts.
5. `PRIME` — The specified files are copied from the staging area to the priming area for use in the final payload. This is distinct from `STAGE` in that the `STAGE` step allows files that are used in the `BUILD` steps of dependent parts to be accessed, while the `PRIME` step occurs after all parts have been staged.

4.9.1 Step order

While each part's steps are guaranteed to run in the order above, they are not necessarily run immediately following each other, especially if multiple parts are included in a project. While specifics are implementation-dependent, the general rules for combining parts are:

1. `PULL` all parts before running further steps.
2. `OVERLAY` parts in their processing order (defined below).
3. `BUILD` any unbuilt parts whose dependencies have been staged. If a part has no dependencies, this part is built in the first iteration.
4. `STAGE` any newly-built parts.
5. Repeat the `BUILD` and `STAGE` steps until all parts have been staged.
6. `PRIME` all parts.

Part processing order

The processing of various parts is ordered based on dependencies. Circular dependencies are not permitted between parts. The ordering rules are as follows:

1. Parts are ordered alphabetically by name
2. Any part that requires another part (using the `after` key) will move that dependency ahead of the declaring part.

NOTE: This means that renaming parts and adding, modifying or removing `after` keys for parts can change the order.

In the example below, the parts will run each stage, ordering the parts alphabetically at each stage (even though C is listed before B):

```
parts:
  A:
    plugin: nil
  C:
    plugin: nil
  B:
    plugin: nil
```

craft_parts output

```
Execute: Pull A
Execute: Pull B
Execute: Pull C
Execute: Overlay A
Execute: Overlay B
Execute: Overlay C
Execute: Build A
Execute: Build B
Execute: Build C
Execute: Stage A
Execute: Stage B
Execute: Stage C
Execute: Prime A
Execute: Prime B
Execute: Prime C
```

However, if parts specify dependencies, both the `build` and `stage` steps of a dependency will be moved ahead of the dependent part in addition to the parts being reordered within a step:

```
parts:
  A:
    plugin: nil
    after: [C]
  C:
    plugin: nil
  B:
    plugin: nil
```

craft_parts output

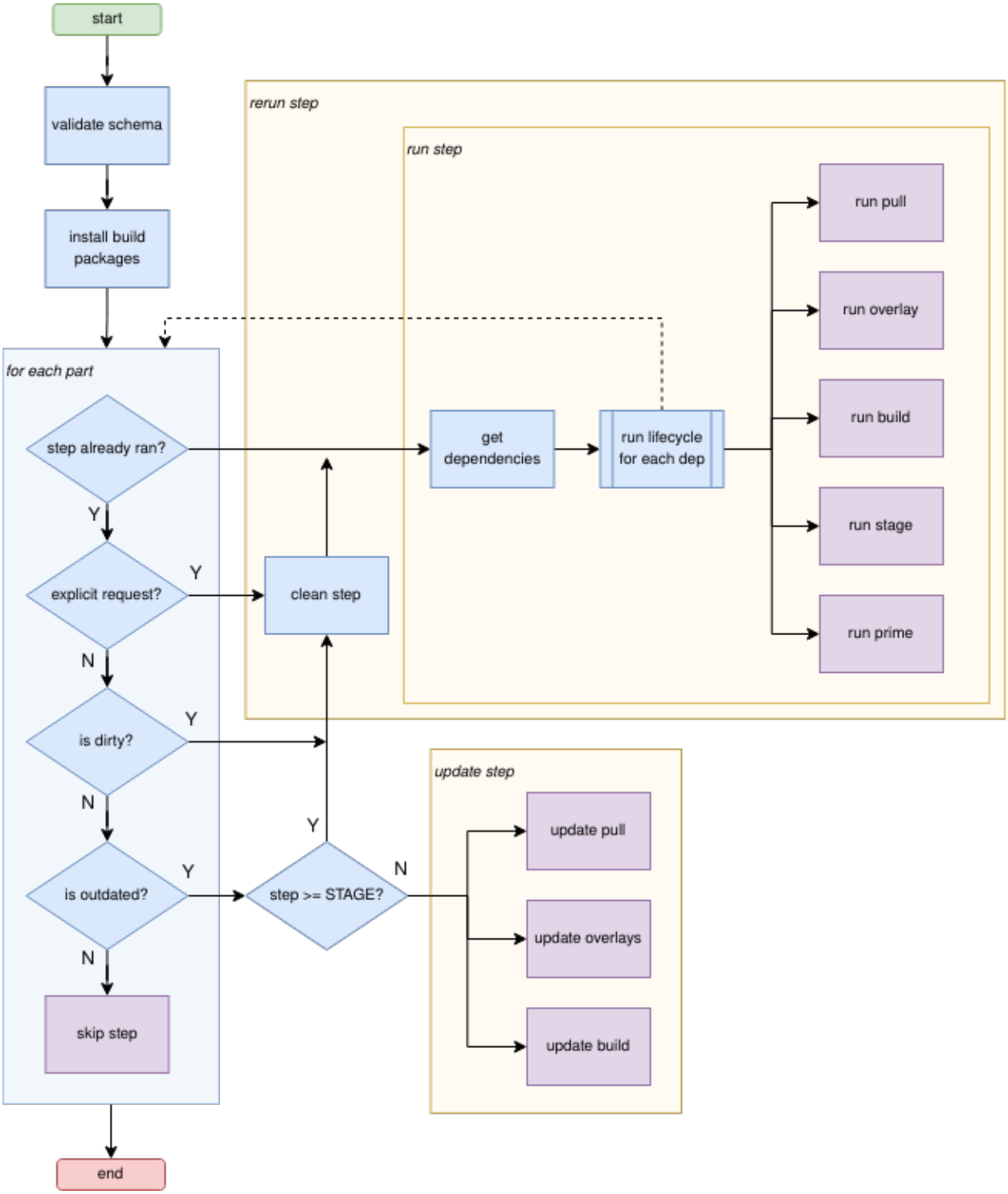
```
Execute: Pull C
Execute: Pull A
Execute: Pull B
Execute: Overlay C
Execute: Overlay A
Execute: Overlay B
Execute: Build C
Execute: Stage C (required to build 'A')
Execute: Build A
Execute: Build B
Execute: Stage A
Execute: Stage B
Execute: Prime C
```

(continues on next page)

(continued from previous page)

```
Execute: Prime A
Execute: Prime B
```

4.9.2 Lifecycle processing diagram



4.9.3 Further Information

Further information can be found in the Snapcraft parts lifecycle documentation.

4.10 Dump Plugin

The dump plugin is a simple plugin that by default is equivalent to running the following command from the source directory:

```
cp --archive --link --no-dereference . "${CRAFT_PART_INSTALL}"
```

The *source* property is the key to this plugin. With some additional properties, they define the location where to get the files from, or which branch, tag, and/or commit to clone if it is a repository.

If you are not using the source files directly in the final payload, but want to run some custom commands to generate them, then you should use the `nil` plugin instead to avoid copying any unnecessary files.

Combining the dump plugin with the *part properties* allows extending the behavior to create new files, modify existing files, and/or filter files for the final payload. For example, it could be used with the *override-build*, to convert file formats.

The plugin can also be used with the *organize* to reorganize the files in the final payload. Like keep only libraries and exclude the binaries and headers from SDKs.

Tutorials **Get started** - become familiar with Rockcraft by containerising different software applications as rocks.

How-to guides **Step-by-step guides** - learn key operations, ranging from *creating and cutting slices* to *migrating and publishing rocks*.

Reference **Technical information** - understand how to use every field in `rockcraft.yaml`.

Explanation **Discussion and clarification** - explore Rockcraft's lifecycle and how a rock gets packed under the hood.

CHAPTER

FIVE

PROJECT AND COMMUNITY

Rockcraft is a member of the Canonical family. It's an open source project that warmly welcomes community projects, contributions, suggestions, fixes and constructive feedback.

- Ubuntu Rocks Discourse
- Rocks Community on Matrix
- Ubuntu Code of Conduct
- Canonical contributor license agreement