CONTENTS

1 Tutorials 3
2 How-to guides 21
3 Reference 37
4 Explanation 63
5 Project and community 75
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.
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.

### 1.1 Create a “Hello World” rock

#### 1.1.1 Prerequisites

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

```yaml
# Metadata section
name: hello
summary: Hello World
description: The most basic example of a rock.
version: "1.0"
license: Apache-2.0

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

(continues on next page)
# Parts section

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

# Parts section

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

This file instructs Rockcraft to build a rock that only has the hello binaries package slice (and its dependencies) inside, using Chisel. 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...  # exported to OCI archive 'hello_1.0_amd64.rock'
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_1.0_amd64.rock'
```

At the end of the process, a file named hello_1.0_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_1.0_amd64.rock docker-daemon:hello:1.0
```

Now run the hello command from the rock:

```
docker run --rm hello:1.0 exec hello -t
```

Which should print:

```
hello, world
```

1.2 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

  _ _ _
| |__ ___| | | ___
| |
\_ \ / _ \ |_/ _ \ |
| | | | __/ | | (_) |
|_| |_|

```

1.2.1 Prerequisites

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

1.2.2 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

### 1.2.3 Pack the rock with Rockcraft

To build the rock, run:

```
rockcraft pack
```

### 1.2.4 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:

```
_ _ _ _ _          ___________   _       _ _______     _ _   _ 
| | | | | __ __ _ __| __ _ _ __| | ___ _| | ___   | | | | | |
| | | | | \/ / _ \/ _ | '_ \/ _ \/ _ \\\\_ -| | | | | | |
| | | | | \//  __/ (_| | | | (_| (_) | | | | | | |
|_| |_| |_|\_\___\___\__\_|_| \______\__\___| |_| |_| |
```

---

Chapter 1. Tutorials
1.2.5 Explore the running container

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

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

1.3 Install slices in a rock

In this tutorial, you will create a lean rock that contains a fully functional OpenSSL installation, and you will verify that it is functional by loading the rock into Docker and using it to validate the certificates of the Ubuntu website.

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)
- Docker installed (https://docs.docker.com/get-docker/)
- a text editor

1.3.2 Install Rockcraft

Install Rockcraft on your host:

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

1.3.3 Project Setup

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

```yaml
name: chisel-openssl
summary: OpenSSL from Chisel slices
description: A "bare" rock containing an OpenSSL installation created from Chisel slices.
license: Apache-2.0

version: "0.0.1"
base: bare
build_base: "ubuntu@22.04"
platforms:
  amd64:
```

(continues on next page)
parts:
openssl:
    plugin: nil
stage-packages:
    - openssl_bins
    - ca-certificates_data

Note that this Rockcraft file uses the openssl_bins and ca-certificates_data Chisel slices to generate an image containing only files that are strictly necessary for a functional OpenSSL installation. See Chisel for details on the Chisel tool.

1.3.4 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 openssl
Executed: overlay openssl
Executed: build openssl
Executed: stage openssl
Executed: prime openssl
Executed parts lifecycle
Exported to OCI archive 'chisel-openssl_0.0.1_amd64.rock'
```

The process might take a little while, but at the end, a new file named chisel-openssl_0.0.1_amd64.rock will be present in the current directory. That’s your OpenSSL rock, in oci-archive format.

1.3.5 Run the rock in Docker

First, import the recently created rock into Docker:

```
sudo /snap/rockcraft/current/bin/skopeo --insecure-policy copy oci-archive:chisel-openssl_0.0.1_amd64.rock docker-daemon:chisel-openssl:latest
```

Now you can run a container from the rock:

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

The output will be OpenSSL’s default help message, which starts like this:

```
help:

Standard commands
asn1parse   ca        ciphers    cmp
           crl       crl2pkcs7  dgst
```

(continues on next page)
As you can see, OpenSSL has many features. Use one of them to check that Ubuntu’s website has valid SSL certificates:

```bash
docker run --rm chisel-openssl exec --env=SSL_CERT_FILE=/etc/ssl/certs/ca-certificates.crt openssl s_client -connect ubuntu.com:443 -brief
```

The output will look similar to the following:

```
CONNECTION ESTABLISHED
Protocol version: TLSv1.3
Ciphersuite: TLS_AES_256_GCM_SHA384
Peer certificate: CN = ubuntu.com
Hash used: SHA256
Signature type: RSA-PSS
Verification: OK
Server Temp Key: X25519, 253 bits
```

The Verification: OK line indicates that the OpenSSL installation inside your rock was able to validate Ubuntu Website’s certificates successfully.

### 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:

```json
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`:

```javascript
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!');
});

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:
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: |
    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

(continues on next page)
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:

```
2023-10-30T12:37:33.659Z [pebble] POST /v1/services 3.878846ms 202
2023-10-30T12:37:33.864Z [app] Service "app" starting: node server.js
```

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


1.5 Migrate a popular Docker image to a chiselled rock

1.5.1 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)
• a text editor

1.5.2 Install Rockcraft

Install Rockcraft on your host:

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

1.5.3 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`:

```bash
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/\n    ->$dotnet_version/dotnet-runtime-$dotnet_version-linux-x64.tar.gz \\
    && dotnet_sha512=\n    ->'c8891b791a51e7d2c3164470dfdf2af2ce59af3c6404e8407527e307df7dcd1e3ccf1a1a32655fe2ee8a8a30f8394b7adbb
    ->' \\
    && 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
```

(continues on next page)
# .NET Runtime version

```bash
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:

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

The output should look as follows:

```
[+] Building 0.6s (10/10) FINISHED
 => [internal] load .dockerignore 0.0s
 => [internal] transfering context: 2B0.0s
 => [internal] load build definition from Dockerfile 0.0s
 => [internal] transfering 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
 => CACHED [stage-1 2/3] COPY --from=installer [/dotnet, /usr/share/dotnet] 0.0s
 => [stage-1 3/3] RUN ln -s /usr/share/dotnet/dotnet /usr/bin/dotnet 0.0s
 => exporting to image 0.0s
 => [stage-1 3/3] writing layers 0.0s
 => [stage-1 3/3] writing image sha256:a24cab51d4d02019dafcd22a2e2a3e1e6d033f9bbf1cb401d465cb2426bb2264 0.0s
 => => naming to docker.io/library/dotnet-runtime:reference 0.0s
```

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

```bash
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:

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

The output should look as follows:

```
global.json file:
Not found
```

(continues on next page)
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

1.5.4 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:

```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:
```

(continues on next page)
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.

1.5.5 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
...
  → install-dotnet-runtime
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))
  → ubuntu 22.04 jammy suite details...
  → for ubuntu 22.04 jammy main component...
```
1.5. Migrate a popular Docker image to a chiselled 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.

### 1.5.6 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
```

(continues on next page)
1.5.7 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.
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:

<table>
<thead>
<tr>
<th>snap</th>
<th>2.57.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>snapd</td>
<td>2.57.1</td>
</tr>
<tr>
<td>series</td>
<td>16</td>
</tr>
<tr>
<td>ubuntu</td>
<td>22.04</td>
</tr>
<tr>
<td>kernel</td>
<td>5.17.0-1016-oem</td>
</tr>
</tbody>
</table>

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

If you're not using Linux, you can still use some of the guides with the appropriate setup.
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](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 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.

2.3 How to contribute to Rockcraft documentation

2.3.1 Tools and markup

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

2.3.2 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.

### 2.3.3 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.

### 2.3.4 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.
2.4 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.

2.5 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.

2.6 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!
2.6.1 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

2.6.2 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:

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

package: <package-name>

# (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: |
      ...
```
2.6.3 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.

2.6.4 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:
```

(continues on next page)
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.

## 2.7 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:
  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 note 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
```
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
ecc param enc engine errstr
gdinstall gendsa genkey genrsa
tool info kdf list
mac nseq ocsp passwd
pks12 pkcs7 pkcs8 pkey
pkpkeyarm pkeyut1 prime rand
rehash req rsa rsautil
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!

2.8 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

```bash
git clone -b ubuntu-22.04 https://github.com/<your_github_username>/chisel-releases.git
```

2. Create a branch:

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

3. Add and commit your modifications:

```bash
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.

## 2.9 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.

### 2.9.1 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.

### 2.9.2 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:

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

(continues on next page)
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).

2.9.3 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: 573BF6DB3D8FBC641079A6ABABF5BD827BD9BF62
      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
      stage-packages:
        - nginx
        - tzdata

(continues on next page)
# Services to be loaded by the Pebble entrypoint

```yaml
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:

```bash
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
Executed: skip stage nginx-user (already ran)
Executed: stage nginx
Executed: stage pebble
Executed: prime nginx-user
```

(continues on next page)
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:

```bash
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:

```bash
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:

```bash
curl localhost:8080
```

For which you should see the following output:

```html
<!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>. Commercial support is available at
<a href="http://nginx.com">nginx.com</a>.<br />
</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```
2.10 Publish a rock to a registry

2.10.1 Prerequisites

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

2.10.2 Push a rock to Docker Hub

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

```
```

Output:

```
Getting image source signatures
Copying blob e65b2e587073 skipped: already exists
Copying blob 01f981dde5a5 skipped: already exists
Copying config 5da2a9016 done
Writing manifest to image destination
Storing signatures
```
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`:

<table>
<thead>
<tr>
<th>organize:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>hello.py</strong>: bin/hello</td>
</tr>
</tbody>
</table>

**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:

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

**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.
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 item
Step: stage
The files to copy from the building area to the staging area.

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.

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.

<table>
<thead>
<tr>
<th>Pull</th>
<th>Build</th>
<th>Stage</th>
<th>Prime</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>after</td>
<td>stage</td>
<td>prime</td>
</tr>
<tr>
<td>source-checksum</td>
<td>build-attributes</td>
<td>stage-snaps</td>
<td></td>
</tr>
<tr>
<td>source-branch</td>
<td>build-environment</td>
<td>stage-packages</td>
<td></td>
</tr>
<tr>
<td>source-commit</td>
<td>build-packages</td>
<td></td>
<td></td>
</tr>
<tr>
<td>source-depth</td>
<td>build-snaps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>source-submodules</td>
<td>organize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>source-subdir</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>source-tag</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>source-type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>override-pull</td>
<td>override-build</td>
<td>override-stage</td>
<td>override-prime</td>
</tr>
</tbody>
</table>

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 | 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.

build-base

Type: One of ubuntu@20.04 | ubuntu@22.04
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.
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.
3.1.3 Example

```plaintext
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
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.

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

--destructorive-mode
  Build in the current host.

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]

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.

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.

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.
```

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.

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.

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 \text{or} --\text{help}
  Show this help message and exit.

-\text{q} \text{or} --\text{quiet}
  Only show warnings and errors, not progress.

-\text{v} \text{or} --\text{verbose}
  Show debug information and be more verbose.

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

-V \text{or} --\text{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 Rockcraft plugins

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

3.3.1 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

By default this plugin uses Python from the base when it is available and appropriate to use. This depends on the tool and format in use.

- The bases used by Rockcraft do not contain Python, so it will need to be supplied in ROCKs that use it.
- Snaps that use strict confinement will use the version of Python in the base. Snaps that use classic confinement will use the host system’s Python.

If a different interpreter is desired, it must be bundled (including the `venv` module) and its path must be included in the `PATH` environment variable.

Use of `python3-<python-package>` in stage-packages will force the inclusion of the Python interpreter.
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.

3.3.2 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.

**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.

**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:

```bash
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:

```bash
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.
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.
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!

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, \( A_{\text{slice3}} \), 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:

```yaml
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:

```yaml
# (...) 
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
```
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*/ symlink. 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)
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.

*Tutorials*  Get started with a hands-on introduction to Rockcraft

*How-to guides*  Step-by-step guides covering key operations and common tasks

*Reference*  Technical information about the *rockcraft.yaml* format

*Explanation*  Discussion and clarification of key topics
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
- Ubuntu Code of Conduct
- Canonical contributor license agreement