Canonical

Ubuntu Packaging Guide

Canonical

Contents

1 Project and community **3**

1.1 Snap Store Proxy How-to guides 3

1.2 Snap Store Proxy Reference 34

The **Snap Store Proxy** provides an on-premise edge proxy to the general Snap Store[1] for your devices.

Devices are registered with the Proxy, and all communication with the Snap Store will **flow through the Proxy**, thereby enabling network-restricted devices to access snaps. Upstream snap **revisions can be overridden** on the Proxy, allowing fine-grained revision control for your devices. The Proxy furthermore supports air-gapped deployments when configured in **offline mode**.

The Proxy is an excellent fit for organisations looking for **more control over updates** to their snaps, or for enterprises that have held back from adopting snaps until now because of the challenges of operating within a **restricted network**.

With the Snap Store Proxy, snaps are as easy-to-use as ever, and administrators have much greater control over exactly what revisions are installed on each connected system.

In this documentation

How-to guides (page 3)

Step-by-step guides covering key operations and common tasks.

Reference (page 34)

Technical information - specifications, APIs, architecture.

[1] https://snapcraft.io/store

1. Project and community

The Snap Store Proxy is a member of the Snap Store family. It's a project that welcomes suggestions, fixes and constructive feedback.

- Get the Snap Store Proxy as a snap[2]
- Join the Discourse forum[3]
- File a bug[4]
- Get support[5]

Thinking about deploying the Snap Store Proxy in your enterprise? Get in touch![6]

1.1. Snap Store Proxy How-to guides

If you have a specific goal, but are already familiar with the Snap Store Proxy, 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.

How-to guides	Get stuff done
Installation (page 4)	Install and set up the Snap Store Proxy
Proxy registration (page 6)	Register the Proxy with the online Snap Store
TLS configuration (page 7)	Configure TLS termination in the Proxy
Configuring snap devices (page 9)	Point your devices to the Proxy instead of the online Snap Store
Overriding snap revisions (page 10)	Control the specific revision of a snap in a channel for your devices
Offline store (page 12)	Deploy the Proxy in an air-gapped environment
Model service (page 18)	Configure an air-gapped Proxy for signing device serial requests
Troubleshooting (page 33)	Check Proxy configuration status and diagnose common issues

Alternatively, our *Tutorials* section contain step-by-step tutorials to help outline what the Proxy is capable of while helping you achieve specific aims.

Take a look at our *Reference* section for technical details (such as the Overrides API specs and authentication mechanism), and other supplementary reference materials.

Finally, for a better understanding of how the Snap Store Proxy works, our *Explanation* section enables you to expand your knowledge.

[2] https://snapcraft.io/snap-store-proxy
[3] https://forum.snapcraft.io/c/store/16
[4] https://bugs.launchpad.net/snapstore-server
[5] https://ubuntu.com/support/community-support
[6] https://ubuntu.com/core/services/contact-us

1.1.1. Installation

Prerequisites

To run the Snap Store Proxy, you will need:

- A server running one of the currently supported LTS versions of Ubuntu[7] on AMD64.
- Firewall rules configured to allow traffic to servers[8].
- A domain name for the server.
- A PostgreSQL instance (see the Database section).

Getting started

First, if your network requires an HTTPS proxy to get to the above domains, you must first configure `snapd` on the installation server to use that HTTPS proxy in order to be able to install the snap-store-proxy snap package.

Do this by adding the appropriate environment variables (`http_proxy`, `https_proxy`) to the server's `/etc/environment` file, and restarting `snapd`:

```
sudo systemctl restart snapd
```

Installing the stable release of the Snap Store Proxy is as simple as:

```
sudo snap install snap-store-proxy
```

This will install the snap, which provides a collection of systemd services, and the `snap-proxy` CLI tool to control the proxy.

Domain configuration

The Snap Store Proxy will require a domain or IP address to be set for the configuration and access by other devices, e.g.:

```
sudo snap-proxy config proxy.domain="snaps.myorg.internal"
```

This can be done after the database is created, but is required before registration can succeed.

[7] https://ubuntu.com/about/release-cycle
[8] https://forum.snapcraft.io/t/network-requirements/5147

Database

When setting up a Snap Store Proxy for production usage, we recommend you have a properly configured PostgreSQL service set up, with backups and possibly HA. However, if you are evaluating the Snap Store Proxy or using it in a local deployment, you can use a local PostgreSQL.

The example below illustrates the expected PostgreSQL set up in terms of a role, database, and a database extension that are required by the Snap Store Proxy.

Example database setup

Ensure that proper PostgreSQL database, user and database extensions are set up. This can be done by adjusting the following script to your needs and running it using `psql` as your PostgreSQL server **superuser**:

```
CREATE ROLE "snapproxy-user" LOGIN CREATEROLE PASSWORD 'snapproxy-password';

CREATE DATABASE "snapproxy-db" OWNER "snapproxy-user";

\connect "snapproxy-db"

CREATE EXTENSION "btree_gist";
```

Simple local Ubuntu setup can look like this:

1. Install PostgreSQL

   ```
   sudo apt install postgresql
   ```

2. Save the above PostgreSQL script as `proxydb.sql` and run it:

   ```
   sudo -u postgres psql < proxydb.sql
   ```

Configure the Snap Store Proxy database

Once the database is prepared, set the connection string:

```
sudo snap-proxy config proxy.db.connection="postgresql://snapproxy-user@localhost:5432/
snapproxy-db"
```

After doing this, you will be prompted to enter the password for that PostgreSQL user.

The connection string format is detailed in the libpq documentation[9].

[9] https://www.postgresql.org/docs/current/static/libpq-connect.html#LIBPQ-CONNSTRING

Network connectivity

You can check that the Proxy can access all the network locations it needs to with:

```
snap-proxy check-connections
```

If you require traffic between your Snap Store Proxy and the internet to go via another HTTP proxy, you can configure your Snap Store Proxy to do so with:

```
sudo snap-proxy config proxy.https.proxy="https://myproxy.internal:3128"
```

Snap Store Proxy also uses the `https_proxy` environment variable if it's set. `http_proxy` is ignored as all outgoing traffic is encrypted.

CA certificates

For verifying outgoing HTTPS communication, Snap Store Proxy bundles a set of root CAs[10] from its base Ubuntu based snap.

On Ubuntu, the system trust store can be modified using `update-ca-certificates` as needed and snap-store-proxy will honour these changes by default (it might require a restart `sudo snap restart snap-store-proxy`).

You can also override this default behaviour and configure your Snap Store Proxy to *only* trust a specific list of CAs:

```
cat your-ca.crt another-ca.crt | sudo snap-proxy use-ca-certs
```

This can be useful in cases when you want your Snap Store Proxy to only trust your internal CA for example.

To reset the CA certificates back to the system defaults, run:

```
sudo snap-proxy remove-ca-certs
```

Next step

Register (page 6) your Snap Store Proxy.

1.1.2. Registration

Initial registration

To register the proxy, you will need to provide Ubuntu SSO credentials for the desired account you wish to link the proxy with, and answer some simple questions about your deployment:

```
sudo snap-proxy register --https
```

or:

[10] https://en.wikipedia.org/wiki/Certificate_authority

```
sudo snap-proxy register
```

If the `--https` option is omitted, the resulting *assertion* (page 9) instructing client devices to use the proxy instead of the upstream store, will instruct them to use HTTP to connect to the proxy instead of HTTPS.

You can examine your proxy's registration status with:

```
snap-proxy status
```

This will show the registration status of your proxy, as well as local status information of this server.

> **Note**
>
> For evaluation purposes, we automatically grant the use of up to 25 devices.

At this point, your proxy will be assigned a Store ID, which can be retrieved with the status command. This will be used in later commands and to identify your proxy for support purposes.

After successful registration it's advised to securely store the private key generated during the process. This key is your proxy's identity. The key pair can be viewed using:

```
sudo snap-proxy config proxy.key.public proxy.key.private
```

Next step

Configure the proxy to *serve HTTPS* (page 7) traffic if `--https` registration option was used, or proceed to *configure client devices* (page 9).

1.1.3. HTTPS

TLS termination is not enabled by default. This means that the proxy listens only on port 80 after installation. If the proxy was *registered* (page 6) with an `--https` option, the resulting *assertion* (page 9) instructing client devices to connect to the proxy instead of the upstream store, is pointing those devices to use HTTPS to connect to the proxy.

This document explains how to enable and configure TLS termination in the Snap Store Proxy.

Certificate and Key

Obtain an x509 key and certificate pair for your Snap Store Proxy domain (as well as any relevant intermediate certificates if applicable). You can determine the domain by running:

```
snap-proxy config proxy.domain
```

This name will be the subject and should be one of the alternative names on the certificate as well.

How to obtain the certificate/key pair is out of scope of this document.

Importing the Key/Certificate pair

Running the below command will import the key/certificate pair (and any intermediate certificates as needed) and re-configure your Snap Store Proxy:

```
cat my.cert my.key [intermediate.cert ...]  | sudo snap-proxy import-certificate
```

For example when configuring a Let's Encrypt issued certificate, you'd want to include the key, the certificate and intermediate certificates. Since Let's Encrypt provides `fullchain.pem` that includes both the site and intermediate certificates, you can use:

```
cat fullchain.pem my.key | sudo snap-proxy import-certificate
```

After this is done, TLS termination will be enabled, and any HTTP traffic to your Snap Store Proxy will be redirected to HTTPS. This command can be re-run as needed.

Self signed certificates

The TLS certificate above may be self signed or ultimately signed by a self signed root CA that is not included in the system certificate store on your client snap devices or the snap-store-proxy host itself. If this is true, then you need to make sure that the self signed certificates in question are added to:

- the system certificate store on the snap-store-proxy host,
- as well as its client devices.

On classic Ubuntu machines this might be achieved by placing the certificate in question in a specific directory:

```
sudo cp my-selfsigned-ca.crt /usr/local/share/ca-certificates/
```

(make sure that the certificate file extension is `.crt`) then running:

```
sudo update-ca-certificates
```

If this is being done on the snap-store-proxy host, the snap-store-proxy has to be restarted:

```
sudo snap restart snap-store-proxy
```

After that, snap-store-proxy will be able to verify its status correctly.

For client machines, `snapd` has to be restarted:

```
sudo systemctl restart snapd
```

After that, `snapd` on the client device will be able to successfully verify the snap-store-proxy certificate.

A more robust method of ensuring that client devices can talk to the snap-store-proxy using a self signed certificate or one issued by a self signed root is to configure the certificate in question using `snapd` itself:

```
sudo snap set system store-certs.cert1="$(cat /path/to/my-cert-or-ca-cert.crt)"
```

The above method works both on classic systems as well as Ubuntu Core.

Next step

Once you've confirmed that your Snap Store Proxy is running and accepting HTTPS connections, you can *configure client devices* (page 9) to use your Snap Store Proxy.

At any time, you can use:

```
snap-proxy status
```

to check the status of your Snap Store Proxy.

1.1.4. Snap devices

Configuring devices

You will need at least `snapd` 2.30 on your device and access to a *registered Snap Store Proxy* (page 6).

To configure `snapd` on a device to talk to the proxy, you need to `snap ack` the signed assertion that allows `snapd` to trust the proxy, e.g.:

```
curl -sL http://<domain>/v2/auth/store/assertions | sudo snap ack /dev/stdin
```

Once `snapd` knows about the store assertion, you then have to configure it to use the proxy:

```
sudo snap set core proxy.store=STORE_ID
```

You can retrieve the `STORE_ID` using the status command on the Proxy server:

```
snap-proxy status
```

Disconnecting devices

If you want to later disconnect a device from the proxy:

```
sudo snap set core proxy.store=''
```

Note that the next time the device refreshes, it will get the upstream snap revisions (any overrides won't be in effect).

Obtaining serial assertions

Devices without a serial assertion[11] are able to obtain one when using the Snap Store Proxy.

If your devices are configured to use a specific `device-service.url` via your gadget snap[12], then `snapd` will send the device registration request to that device service via the Snap Store Proxy. This means that you can use a specific serial-vault service to obtain serial assertions for your devices running behind a Snap Store Proxy. Make sure that your Snap Store Proxy is able to connect to this specific serial-vault service.

!!! NOTE: By default Snap Store Proxy allows only the `https://serial-vault-partners.canonical.com` serial-vault requests to pass through it.

[11] https://docs.ubuntu.com/core/en/reference/assertions/serial
[12] https://snapcraft.io/docs/gadget-snap

Since version 2.19 of the snap-store-proxy, the `proxy.device-auth.allowed-device-service-urls` setting can be used to control the list of allowed device services (Serial Vaults), e.g.:

```
sudo snap-proxy config \
    proxy.device-auth.allowed-device-service-urls='["https://sv1.internal", "https://sv2.internal"]'
```

Next step

With devices connected to the proxy, you can *create overrides* (page 10) to control snap updates on them.

1.1.5. Snap revision overrides

You can override the revisions for specific snaps, on a specific channel[13]. This means you can control the specific revision of a snap in a channel, rather than what the upstream publisher has released. You can use this to effectively pin revisions, and control when you are ready to upgrade to newer revisions.

There are a few different ways to configure overrides.

Proxy server

To configure overrides from the proxy server, use the `snap-proxy` command.

To add an override:

```
sudo snap-proxy override <snap> <channel>=<revision>
```

To list overrides currently in place:

```
sudo snap-proxy list-overrides <snap>
```

To remove all current overrides on a channel:

```
sudo snap-proxy delete-override <snap> <channel>
```

Revisions and Architectures

A Snap Store channel can publish only one revision[14] of a specific snap at any time.

A snap revision can support one or multiple architectures. Specifying a revision for an override therefore also determines which architectures the override is set for.

Revisions for specific snaps can be looked up using the `snap info` command, which lists currently available revisions for the architecture of the device running this command. Snap Stores Devices API snaps_info[15] endpoint can also be used to obtain available revisions for selected architectures.

[13] https://docs.snapcraft.io/reference/channels
[14] https://snapcraft.io/docs/getting-started
[15] https://api.snapcraft.io/docs/info.html

In the example below, we have the `core18` snap and two revisions, each supporting one architecture.

```
# 1722 is one of the amd64 revisions of the core18 snap.
$ sudo snap-proxy override core18 stable=1722
core18 stable amd64 1722

# 1725 is one of the armhf revisions of the core18 snap.
$ sudo snap-proxy override core18 stable=1725
core18 stable armhf 1725

# We can see that we've overriden the stable channel revisions for both
# amd64 and armhf and that both upstream counterparts ar at lower revisions.
$ sudo snap-store-proxy list-overrides core18
core18 stable amd64 1722 (upstream 1705)
core18 stable armhf 1725 (upstream 1706)

# Deleting a channel-specific override deletes overrides for all revisions
# and architectures.
$ sudo snap-store-proxy delete-override core18 stable
core18 stable amd64 is tracking upstream (revision 1705)
core18 stable armhf is tracking upstream (revision 1706)
```

Overrides API

Alternatively, you can also manage overrides via a *REST API* (page 35)

Command line tool

There is a CLI tool[16] to help manage overrides, which uses the API and can be used remotely to administer overrides:

```
sudo snap install snapstore-client
```

Authentication is performed using Ubuntu SSO, and users need to be authorised from the CLI on the server using:

```
sudo snap-proxy add-admin becky@example.com
```

On the client side, you authenticate by:

```
snapstore-client login
```

Overrides are managed in the same way as with the `snap-proxy` command above, e.g.:

```
snapstore-client list-overrides
snapstore-client override foo stable=10
snapstore-client delete-override foo
```

[16] https://snapcraft.io/snap-store-proxy-client

1.1.6. Offline store (air-gapped mode)

By default, the Snap Store Proxy operates in online mode. It acts as a smart proxy to the general SaaS Snap Store.

Snap Store Proxy can operate in offline mode and act as a local Snap Store (on-prem store), meaning it can be deployed in networks that are disconnected from the internet.

The intended use case for this mode are network restricted environments where no outside traffic is allowed or possible.

Overview

Client devices connect to the offline store. The local store doesn't directly contact the general SaaS Snap Store nor the internet.

Proxy operators side-load all necessary snaps and other metadata into their local store by exporting them from the SaaS store first and then importing into their offline store.

Brand Store support

Brand Store[17] (also known as IoT App Store[18]) customers can use the Snap Store Proxy in offline mode to securely serve updates to their fleet of devices.

Operators can import their brand store snaps (including any essential and other snaps included from the global store in their brand store) to their on-prem store.

Devices with valid serial assertions for models belonging to a specific brand can authenticate to such on-prem store and get access to imported brand store snaps which are not accessible to any other devices connecting to that on-prem store.

> **Note**
>
> Client devices have to be equipped with their *serial assertions* (page 9).

Installation

If the target host has internet access at the time of installation, then the regular *installation* (page 4) and *registration* (page 6) can be used followed by airgap mode activation:

```
sudo snap-proxy enable-airgap-mode
```

> **Note**
>
> Even though it's possible to enable airgap mode for an online proxy, it's only advised to do so during the installation phase when no devices are yet *connected* (page 9) to the proxy. Deactivating and activating airgap mode while it's already serving clients will have

[17] https://ubuntu.com/core/docs/store-overview
[18] https://ubuntu.com/internet-of-things/appstore

undesirable and not clearly defined consequences for the devices that were connected to it before the mode switch as well as for the snap-store-proxy instance itself.

Offline installation

To deploy an offline store (Snap Store Proxy in air-gapped mode) to a machine without internet access it's possible to register it using a separate machine first and install the resulting package on the target machine.

Air-gapped Snap Store Proxy operators first have to register their offline proxy on a **machine with internet access**. This can be done using the `store-admin` snap:

```
sudo snap install store-admin
```

On the same machine, register the store and obtain a tarball for installation on an offline host (partly pre-configured as well):

```
# You'll be prompted to authenticate with your Ubuntu SSO authentication.
store-admin register --offline <target-http-location-of-the-store>
```

Warning

```
Full value of the target location, eg `https://snaps.internal`, will be encoded
in an assertion file used for instructing client devices to connect to this
store. It's important to decide if http or https will be used and what the host
name will be at the point of registration.
```

The result of the above is a tarball `offline-snap-store.tar.gz` that is then moved to the target host machine for the offline store for installation.

The target machine (the air-gapped Snap Store Proxy host) should have network access to a *properly configured PostgreSQL database* (page 5).

You will need the `offline-snap-store.tar.gz` bundle from the registration step to continue with installation.

The script below illustrates the installation process on the target air-gapped machine. Please note that the following variables need to be set appropriately:

- `POSTGRESQL_CONN_STRING` - the connection string to a *properly set up PostgreSQL database* (page 5)

```
#!/bin/bash

set -eu

# PostgreSQL connection string to the Snap Store Proxy database.
POSTGRESQL_CONN_STRING="${POSTGRESQL_CONN_STRING}"

tar xvzf offline-snap-store.tar.gz
cd offline-snap-store
sudo ./install.sh
```

(continues on next page)

(continued from previous page)

```
sudo snap-store-proxy config proxy.db.connection="$POSTGRESQL_CONN_STRING"

sudo snap-store-proxy enable-airgap-mode

sudo snap-store-proxy status
```

If the registered store's location was an HTTPS one, follow the *HTTPS setup* (page 7) instructions to configure the TLS certificate.

Brand store metadata import

Warning

```
This section is relevant for brand store customers wanting to host their
brand store snaps offline and can be skipped if the offline store only has
to support Global store client devices (eg. generic devices).
```

On-prem stores need various data (assertions, snap binaries and metadata, account information) - produced by the upstream Snap Store - to function. This data has to be exported from the SaaS IoT App store and imported into the on-prem store at least once.

Any account keys[19] used for signing brand devices' models[20] and serials[21] have to be registered with the SaaS Snap Store using `snapcraft register-key` (by the brand account) prior to the export in order for the on-prem store to be able to authenticate brand store devices.

The store export and import steps can be repeated to "synchronise" the data and/or snaps from the SaaS store as needed.

Brand store export

To export brand store metadata needed for import to the on-prem store, the `store-admin export store` command can be used on a machine with internet access. Authentication using an account with *Admin* role for the brand store in question is required. Example:

```
$ store-admin export store \
    --arch=amd64 --arch=arm64 \
    --channel=stable --channel=edge \
    --key=keyId1 --key=keyId2 \
    myDeviceViewStoreID

Logging in as store admin...
Opening an authorization web page in your browser.
If it does not open, please open this URL:
 https://api.jujucharms.com/identity/login?did=idxyz
```

(continues on next page)

[19] https://ubuntu.com/core/docs/reference/assertions/account-key
[20] https://ubuntu.com/core/docs/reference/assertions/model
[21] https://ubuntu.com/core/docs/reference/assertions/serial

(continued from previous page)

```
Downloading store metadata and assertion...
Downloading store admin account details and assertion...
Downloading snap declaration for my-registered-unbpublished-snap1...
Downloading account-key keyId1...
Downloading account-key keyId2...
Downloading core revision 13250 (latest/stable amd64)
  [####################################]  100%
Downloading core revision 13253 (latest/stable arm64)
  [####################################]  100%

...

Creating the export archive...
Store data exported to: /home/ubuntu/snap/store-admin/common/export/store-export-
myDeviceViewStoreID-20220527T082652.tar.gz
```

The above will export the following data:

- SaaS IoT App Stores' (device view store and its parent) metadata,
- Registered public keys in form of account-key assertions for key IDs specified with the `--key` option. Make sure to include the keys used for signing client device serial and model assertions. These keys have to be registered using `snapcraft register-key` command prior to the export, by the brand account.
- Snaps available in the SaaS stores, with their metadata and assertions. Currently published revisions of the snaps will be exported according to the specified architectures and channels: `--arch`, `--channel`. The `--no-snaps` option can be used to skip the export of any snap revisions (`store-admin export snaps` can be used to export snaps in a more granular fashion).

Brand store import

The exported `store-export-*.tar.gz` file can be imported on the target on-prem host using the `snap-proxy push-store` command. Example:

```
sudo snap-proxy push-store \
    /var/snap/snap-store-proxy/common/snaps-to-push/store-export-myDeviceViewStoreID.tar.
gz
```

Side-loading snaps

It's possible to export snaps from the upstream Snap Store and import them into their on-prem store. These will be the only snaps (and their revisions) available for installation from the on-prem store.

Exporting snaps

Example of exporting `jq` and `htop` snaps on a **machine with internet access** using the `store-admin` snap:

```
$ store-admin export snaps jq htop --channel=stable --arch=amd64 --arch=arm64 --export-dir
.
Downloading jq revision 6 (latest/stable amd64)
  [####################################]  100%
Downloading jq revision 8 (latest/stable arm64)
  [####################################]  100%
Downloading htop revision 3417 (latest/stable amd64)
  [####################################]  100%
Downloading htop revision 3425 (latest/stable arm64)
  [####################################]  100%
Successfully exported snaps:
jq: jq-20221026T104628.tar.gz
htop: htop-20221026T104628.tar.gz
```

This produces a set of `tar.gz` files (one per snap name) that have to be moved to the on-prem store host and imported there.

> **Note**
>
> By default snaps are exported from the Global store, and then imported as such, meaning that any device connected to the on-prem store will be able to install them (if it's configured to use the default Global store). `store-admin export snaps` has a `--store` option allowing for authenticated export of snaps from private device-view IoT App Stores - after importing these, snaps will be accessible only to properly authenticated devices from the relevant brand.

Importing (pushing) snaps

Once the snap bundles are on the on-prem store host, they should be moved to the `/var/snap/snap-store-proxy/common/snaps-to-push/` directory, from where they can be imported.

Example of importing a `jq.tar.gz` snap bundle on the air-gapped proxy host:

```
sudo snap-store-proxy push-snap /var/snap/snap-store-proxy/common/snaps-to-push/jq-
20200406T103511.tar.gz
```

The `jq` snap is now available for installation from this air-gapped Snap Store Proxy. This means that `snap info jq` and `snap install jq` will succeed on a connected client device.

Essential snaps

Devices that connect to a store expect that snaps pre-installed on those devices will be available in that store. Otherwise common operations like `snap refresh` will fail. Air-gapped store operators should ensure that all necessary snaps are imported. Snaps commonly pre-installed on devices may include but are not limited to:

- `core`
- `core18`
- `core20`
- `core22`
- `snapd`

Status

```
snap-store-proxy status
```

lists the imported stores and account keys and

```
snap-store-proxy list-pushed-snaps
```

lists all imported snaps.

Running `snap info <snap-name>` from a device connected to the on-prem store can be used to view more details about the snap, like its current channel map.

Client Device Configuration

Configuring client devices (page 9) follows the same process as with an online Snap Store Proxy.

Offline Upgrades

To upgrade snap-store-proxy on an offline machine, first download the snap and its assertions on a machine with internet access, e.g.:

```
snap download snap-store-proxy --channel=latest/stable
```

Same can be done for its base snap `core22` and for the `snapd` snap itself.

Then move the files over to the offline snap-store-proxy machine and:

```
sudo snap ack snap-store-proxy_<revision>.assert
sudo snap install snap-store-proxy_<revision>.snap
```

And use analogous process to upgrade the base and `snapd` snaps.

Configuration backup

Make sure to securely backup the snap-store-proxy configuration (including the `proxy.device-auth.secret` used for signing/verifying the device sessions). The configuration can be exported with:

```
sudo snap-store-proxy config > proxy-config-backup.txt
sudo snap-store-proxy config proxy.device-auth.secret > proxy.device-auth.secret.txt
sudo snap-store-proxy config proxy.auth.secret > proxy.auth.secret.txt
sudo snap-store-proxy config proxy.key.private > proxy.key.private.txt
sudo snap-store-proxy config proxy.tls.key > proxy.tls.key.txt
```

Limitations

Offline mode provides only a subset of the core functionality of the online Snap Store Proxy or the SaaS Snap Store. Some of the missing features are:

- Searching for snaps
- Generic Device registration. Serial Vault can be used to register custom model devices.

1.1.7. On-Prem Model Service

The Snap Store's Model Service is the device serial provisioning service that is intended to supersede the Serial Vault[22]. While the Serial Vault had to be deployed separately in on-prem scenarios, the Model Service is packaged into the Snap Store Proxy.

The Model Service is currently only available in *air-gapped* (page 12) mode. When operating in online mode, the Proxy can forward serial requests made by devices to the online Serial Vault or the online Model Service.

The following requirements need to be met to use the on-prem Model Service:

- A PKCS#11-compatible Hardware Security Module (HSM) or Smart Card.
- The PKCS#11 module / shared library for the hardware. e.g. `opensc-pkcs11.so`
- The Snap Store Proxy host machine must run Ubuntu 22.04 (Jammy) with the `p11-kit` and `gnutls-bin` packages installed.
- Revision 99 of `snap-store-proxy` and revision 28 of `store-admin`, or newer.

The supported way to manage models, signing keys, and serial policies in the on-prem Model Service is via the `store-admin` snap.

[22] https://ubuntu.com/core/services/guide/serial-vault-overview

HSM-based key management and signing

Currently, only PKCS#11-compatible hardware can be used for key generation and signing device serial requests. Software key generation and signing are not supported, with the exception of using a software PKCS#11 emulator such as SoftHSMv2[23].

p11-kit[24] is used as an abstraction layer for hardware-agnostic PKCS#11 support.

Setup

This section assumes that the Snap Store Proxy has been installed and configured in air-gapped mode and that the Brand Store has been imported, as per the installation steps in the *Offline store section* (page 12). On Brand Store import, the Brand account and admin user will be automatically set up in the Model Service.

Store admin token

To login using `store-admin` to the Model Service, set the `STORE_ADMIN_TOKEN` environment variable, obtained after running `store-admin export store`:

```
$ store-admin export store myDeviceViewStoreID --key <model-assertion-account-key-sha3-
384>

...

Creating the export archive...
Store data exported to: /home/ubuntu/snap/store-admin/common/export/store-export-
myDeviceViewStoreID-20240109T123041.tar.gz
Admin token exported to: /home/ubuntu/snap/store-admin/common/export/store-export-
myDeviceViewStoreID-20240109T123041.macaroon
Admin token usage:
    export STORE_ADMIN_TOKEN=$(cat /home/ubuntu/snap/store-admin/common/export/store-
export-myDeviceViewStoreID-20240109T123041.macaroon)
```

As outlined in the *air-gapped store setup instructions* (page 14), the account-key assertion for the key(s) used to sign the model assertion(s)[25] must also be exported and pushed to the Proxy. Include them in the store export bundle by specifying the `--key` flag for each account-key SHA3-384.

Import the store bundle on the Proxy (page 15), then login to the air-gapped store from the admin machine:

```
$ store-admin login --offline <http-location-of-the-store> <same-email-as-in-export-store>

Exchanging store admin macaroon for a publisher gateway admin macaroon...
```

Access via the `store-admin` snap should now be set up for the Model Service.

[23] https://github.com/opendnssec/SoftHSMv2
[24] https://p11-glue.github.io/p11-glue/p11-kit.html
[25] https://ubuntu.com/core/docs/sign-model-assertion

p11-kit server

On the Proxy host, start the p11-kit server.

Obtain the `pkcs11:` identifier using `p11tool`, e.g.:

```
$ p11tool --provider "/usr/lib/x86_64-linux-gnu/opensc-pkcs11.so" --list-token-urls | sed
's/;token=.*//g'

pkcs11:model=PKCS%2315%20emulated;manufacturer=www.CardContact.de;serial=DENK0300972
```

Start the server, ensuring that the Unix socket runs under `/var/snap/snap-store-proxy/common/pkcs11`. See the p11-kit documentation[26] for other configuration options.

```
$ sudo p11-kit server --provider /usr/lib/x86_64-linux-gnu/opensc-pkcs11.so
"pkcs11:model=PKCS%2315%20emulated;manufacturer=www.CardContact.de;serial=DENK0300972" -n
"/var/snap/snap-store-proxy/common/pkcs11" -f

P11_KIT_SERVER_ADDRESS=unix:path=/var/snap/snap-store-proxy/common/pkcs11; export P11_KIT_
SERVER_ADDRESS;
P11_KIT_SERVER_PID=26963; export P11_KIT_SERVER_PID;
```

Restart the Model Service (this needs to be done each time the p11-kit server is restarted):

```
snap restart snap-store-proxy.snapmodels
```

Snap Store Proxy configuration

Set the HSM label and pin in the Proxy snap:

```
$ p11tool --list-tokens

Token 0:
    Label: SmartCard-HSM (UserPIN)
...

$ sudo snap-proxy config proxy.hsm.token-label="SmartCard-HSM (UserPIN)"

$ sudo snap-proxy config proxy.hsm.token-pin=74656
```

Model Service CLI Usage

The Model Service management CLI is provided by the `store-admin` snap.

[26] https://p11-glue.github.io/p11-glue/p11-kit/manual/

Create a signing key on the HSM

Create a signing key using `store-admin` for signing serial requests:

```
$ BRAND_ACCOUNT_ID=<brand-account-id> store-admin create key test-key
Generating a signing keypair on the proxy's HSM. This may take some time.
Signing key 'test-key' created.

$ store-admin list keys
Name      SHA3-384
--------  -----------------------------------------------------------
test-key  PPkB6XcYjkxzA9c6dXsaM0sg9r_d5DZ2kDYvWPTeuSXofXGzMDBt7DoD_Xiw3see
```

The `BRAND_ACCOUNT_ID` environment variable only needs to be set once; it will be stored and automatically used subsequently.

> **Note**
>
> If a 4096-bit RSA key takes more than 15 seconds to generate on your hardware (e.g. Nitrokeys), then you would first have to extend the Proxy's internal service timeout: `sudo snap-store-proxy config internal.publishergw.snapmodels.read-timeout={timeout-in-seconds}`

The key needs to be registered with the online Snap Store before it can sign serials:

```
$ store-admin register-key PPkB6XcYjkxzA9c6dXsaM0sg9r_d5DZ2kDYvWPTeuSXofXGzMDBt7DoD_
Xiw3see
Registering signing key with the global Snap Store...
...

Key PPkB6XcYjkxzA9c6dXsaM0sg9r_d5DZ2kDYvWPTeuSXofXGzMDBt7DoD_Xiw3see registered.
```

The account-key assertion needs to be pushed to the air-gapped Proxy. First, export the assertion:

```
snap known --remote account-key public-key-sha3-384=PPkB6XcYjkxzA9c6dXsaM0sg9r_
d5DZ2kDYvWPTeuSXofXGzMDBt7DoD_Xiw3see > test-key.assert
```

Copy the assertion to the Snap Store Proxy's `$SNAP_COMMON` directory on the Proxy host, then push the assertion to the Proxy:

```
sudo snap-proxy push-account-keys /var/snap/snap-store-proxy/common/test-key.assert
```

> **Note**
>
> Repeat these steps to add new account-keys to the proxy, if any are created after the initial store import and are used to sign new model assertions.

Add a model in the Model Service

To sign the serial requests for devices of a model, the model name as configured in the Model Service must match that in the model assertion:

```
$ store-admin create model model-a
API key (alphanumeric, `pwgen 40` for options): model-a-apikey
Model 'model-a' created.
```

Configure a serial signing policy

The serial policy for a model identifies the signing key that the Model Service should use to sign serial requests for that model. To configure the Model Service to sign `model-a` device serial requests with the earlier-created `test-key`:

```
$ store-admin create serial-policy
Model to attach the serial signing policy: model-a
Signing key to use (SHA3-384): PPkB6XcYjkxzA9c6dXsaM0sg9r_d5DZ2kDYvWPTeuSXofXGzMDBt7DoD_
Xiw3see
Model 'model-a' configured to sign serials with key 'PPkB6XcYjkxzA9c6dXsaM0sg9r_
d5DZ2kDYvWPTeuSXofXGzMDBt7DoD_Xiw3see'.

$ store-admin list models
Name     API key         Active serial signing key
-------  --------------  --------------------------
model-a  model-a-apikey  test-key
```

The key can be changed by creating a new serial policy revision for the model:

```
$ store-admin create key new-key
...

$ store-admin list keys
Name      SHA3-384
--------  -----------------------------------------------------------------
test-key  PPkB6XcYjkxzA9c6dXsaM0sg9r_d5DZ2kDYvWPTeuSXofXGzMDBt7DoD_Xiw3see
new-key   4aPBeXLPu2xoriNr-6e1Ja448wC7IAS86Ijs6r0sHWiZ6Y9WYprxeWkK7HETCyh6

$ store-admin create serial-policy
Model to attach the serial signing policy: model-a
Signing key to use (SHA3-384): 4aPBeXLPu2xoriNr-
6e1Ja448wC7IAS86Ijs6r0sHWiZ6Y9WYprxeWkK7HETCyh6
Model 'model-a' configured to sign serials with key '4aPBeXLPu2xoriNr-
6e1Ja448wC7IAS86Ijs6r0sHWiZ6Y9WYprxeWkK7HETCyh6'.

$ store-admin list models
Name     API key         Active serial signing key
-------  --------------  --------------------------
model-a  model-a-apikey  new-key
```

Device gadget snap

The `prepare-device` hook in the device gadget[27] snap should be configured with the proxy host URL and the model API key defined in the Proxy Model Service.

On first startup, a `model-a` device should request and obtain a serial assertion from the Snap Store Proxy:

```
$ snap model --serial --assertion

type: serial
...
model: model-a
...
sign-key-sha3-384: PPkB6XcYjkxzA9c6dXsaM0sg9r_d5DZ2kDYvWPTeuSXofXGzMDBt7DoD_Xiw3see
...
```

1.1.8. Charmhub Proxy

Version 3.0 (and above) of the Snap Store Proxy snap is able to serve Charms and Charm Bundles in air-gapped Juju deployments. We refer to these functionalities as the Charmhub Proxy (CHP). The overall workflow is similar to that for the *offline snap store* (page 12): export, push, then deploy.

Setup

Follow the steps in the *Installation* (page 12) section to register an offline store and pack an offline store installation bundle.

Export packages

On an internet-connected machine, export the required charms and resources as illustrated in the following sections, then copy the exported files to the air-gapped Charmhub Proxy host (or registry host for OCI images).

Export charm bundles

If the offline deployment target is a Charmhub bundle[28], then the bundle and its component charms can be exported like so:

```
$ store-admin export bundle cos-lite --channel=latest/stable --series=kubernetes --
arch=amd64
Downloading cos-lite revision 11 (stable)
  [####################################]  100%
Downloading traefik-k8s revision 176 (stable)
...
...
Successfully exported charm bundle cos-lite: /home/ubuntu/snap/store-admin/common/export/
cos-lite-20240426T143758.tar.gz
```

[27] https://ubuntu.com/core/docs/gadget-snaps#heading--example-prepare
[28] https://charmhub.io/?type=bundle

If not specified, `channel` defaults to `<default-track>/stable`, `series` defaults to `jammy`, and `arch` defaults to `amd64`. If a charm is not found for a given Ubuntu series, the exporter will attempt to fallback or fall forward to the release available for the nearest LTS.

The component charms in the bundle will be auto-exported based on the channel and series defined in the bundle. If it is necessary to modify the bundle before exporting the component charms, then the `bundle.yaml` can be printed to `stdout` by passing `--only-print-yaml=True`.

Export charms

To export individual charms, either start from an existing `bundle.yaml` or define one with a list of cherry-picked charms. The following shows an example for a custom-defined `charms.yaml` (can be any file name):

```
applications:
  postgresql:
    charm: postgresql-k8s
    series: kubernetes
    channel: latest/stable
    resources:
      postgresql-image: postgres:local-image
  kafka:
    charm: kafka-k8s
```

The `charm` key is required, while the other fields will use default values if omitted. All extra keys will be ignored. The `resources` key is only necessary if a manual override of the OCI image subpath is required. See *Export OCI images* (page 26) for use cases.

Pass the `charms.yaml` to the `export charms` command like so:

```
$ store-admin export charms ./charms.yaml
Overriding postgresql-image with local registry subpath.
Downloading postgresql-k8s revision 20 (latest/stable)
  [########################################]  100%
Downloading resources for postgresql-k8s
...
Successfully exported charms to: /home/ubuntu/snap/store-admin/common/export/charms-
export-20240426T163115.tar.gz
```

The exported `tar.gz` contains the following:

```
charms-export-20240426T163115.tar.gz/
├─ bundle.yaml
├─ postgresql-k8s.tar.gz/
|  ├─ postgresql-k8s_20.charm
|  ├─ postgresql-k8s_publisher_account.assert
|  ├─ metadata.json
|  ├─ resources/
|  |  ├─ postgresql-k8s.postgresql-image_19
```

where:

- `metadata.json` is the charm metadata fetched, this is required by CHP to write the charm's channel map and other metadata to enable deployments.
- `bundle.yaml` is a copy of the user-specified export `.yaml`, provided for ease of reference.

- `postgresql-k8s_20.charm` is the binary downloaded from Charmhub, for the revision resolved.
- `postgresql-k8s_publisher_account.assert` is the account assertion of the charm publisher.
- `resources/` contain resource binaries attached to the exported charm revision.

Export snap resources

Some charms may require a specific snap revision as a resource. These charms usually run the equivalent of `snap install <snap> --revision <rev>` in their setup code (example[29]). To export snaps by revision, define a `.yaml` file of the following structure:

```
packages:
  - name: charmed-postgresql
    revision: 96
    push_channel: chp_14/edge
  - name: charmed-mysql
    revision: 97
    push_channel: 8.0/edge
```

The Snap Store implementation of snap installs by revision requires that the revision exists in the snap's channel map history, i.e. the revision must have been released to any channel before it can be requested directly. Thus, `push_channel` needs to be specified to tell Snap Store Proxy the target channel for the revision. This can be a channel that exists for the snap, thereby effectively overriding the channel when the snap is pushed, or it can be an arbitrary track, which would be created in the Proxy on push.

The export `.yaml` can be supplied to the `export snaps` command like so:

```
$ store-admin export snaps --from-yaml snaps.yaml
Downloading charmed-postgresql revision 96 (chp_14/edge amd64)
  [####################################]  100%
Downloading charmed-mysql revision 97 (8.0/edge amd64)
  [####################################]  100%
Successfully exported snaps:
charmed-postgresql: /home/ubuntu/snap/store-admin/common/export/charmed-postgresql-
20240429T122503.tar.gz
charmed-mysql: /home/ubuntu/snap/store-admin/common/export/charmed-mysql-20240429T122503.
tar.gz
```

[29] https://github.com/canonical/postgresql-operator/blob/9614915048ba612bb4be6a5fd8c752a46bb051ed/lib/charms/operator_libs_linux/v2/snap.py#L460

Export OCI images

A local OCI registry needs to be set up to enable charms with OCI image resources. On charm export, the OCI image metadata blob is written to the `resources` directory, e.g. for `postgresql-k8s`:

```
{
  "ImageName": "registry.jujucharms.com/charm/kotcfrohea62xreenq1q75n1lyspke0qkurhk/
postgresql-image@sha256:8a72e1152d4a01cd9f469...",
  "Password": "MDAxOGxvY2F0aW9uIGNoYXJtc3Rvcm...",
  "Username": "docker-registry"
}
```

The image itself needs to be exported using a separate tool such as `skopeo`, which can transfer images between registries, and between a registry and a local directory, while preserving the image hash.

For example, to save the above image to a local directory:

```
$ skopeo copy docker://registry.jujucharms.com/charm/
kotcfrohea62xreenq1q75n1lyspke0qkurhk/postgresql-image@sha256:8a72e1152d4a0... --src-
creds=docker-registry:MDAxOGxvY2F0aW9... dir:/home/ubuntu/<target-dir>
```

The directory can then be manually copied to the air-gapped registry host, then pushed to the registry like so:

```
$ skopeo copy dir:/home/ubuntu/<copied-dir> docker://<local-registry-domain>/charm/
kotcfrohea62xreenq1q75n1lyspke0qkurhk/postgresql-image@sha256:8a72e1152d4a0... --dest-
creds=<local-registry-username>:<local-registry-password>
```

By default, if no override is supplied via the `resources` key in the `.yaml` supplied for charm export, Charmhub Proxy will assume an identical local registry image path (excluding the domain but including `charm/` and including the sha256 tag). When a deployment is requested, CHP will supply a regenerated blob using the local domain URL and credentials configured.

The `skopeo` commands above pushes the image to the same path in the local registry and saves the effort of manually remapping resources. If required, the image can be pushed to a custom path, but a mapping must be defined for the resource as in the example `charms.yaml` in *Export charms* (page 24).

Offline Charmhub (air-gapped mode)

Version 2.30 (and above) of the Snap Store Proxy snap can also distribute charms and charm bundles in addition to snaps. Currently, this functionality is only available in offline mode, enabling the proxy to act as a local (on-premise) Charmhub.

This mode is particularly useful in network-restricted environments where external internet traffic is either not allowed or not possible.

The scope of this configuration is strictly limited to Charmhub-related activities, encompassing the management of charms, snaps, and their related resources. It does not cover the handling of other resources such as APT packages or any custom dependencies that a charm might require during runtime.

The Charmhub proxy does not incorporate an OCI registry. Users who work with Kubernetes charms must establish their own local OCI registry to manage container images.

Offline Charmhub Configuration

Once the user has completed the airgap *Store proxy installation* (page 12), the only remaining step is to configure the path to their local OCI registry.

Configure OCI registry

To configure your local OCI registry, specify a domain name or an IP. This setting should omit the protocol, requiring only the domain name itself.

In this setup, the local Charmhub does not directly access the registry; rather, it provides the image path and credentials to Juju. Juju then uses this information to instruct the container runtime to fetch the images.

The domain name and credentials are configured to override the default upstream domain and credentials, ensuring that charm OCI image paths are correctly served from your local setup.

```
sudo snap-store-proxy config proxy.oci-registry.domain=<registry-domain>
```

If required, set the credentials for registry access.

```
sudo snap-store-proxy config proxy.oci-registry.username=some-username proxy.oci-registry.
password=some-password
```

Export packages

On an internet-connected machine, export the required charms and resources as illustrated in the following sections, then copy the exported files to the air-gapped Charmhub Proxy host (or registry host for OCI images).

Export charm bundles

If the offline deployment target is a Charmhub bundle[30], then the bundle and its component charms can be exported like so:

```
$ store-admin export bundle cos-lite --channel=latest/stable --series=kubernetes --
arch=amd64
Downloading cos-lite revision 11 (stable)
  [####################################]  100%
Downloading traefik-k8s revision 176 (stable)
...
...
Successfully exported charm bundle cos-lite: /home/ubuntu/snap/store-admin/common/export/
cos-lite-20240426T143758.tar.gz
```

[30] https://charmhub.io/?type=bundle

If not specified, `channel` defaults to `<default-track>/stable`, `series` defaults to `jammy`, and `arch` defaults to `amd64`. If a charm is not found for a given Ubuntu series, the exporter will attempt to fallback or fall forward to the release available for the nearest LTS.

The component charms in the bundle will be auto-exported based on the channel and series defined in the bundle. If it is necessary to modify the bundle before exporting the component charms, then the `bundle.yaml` can be printed to `stdout` by passing `--only-print-yaml=True`.

Export charms

To export individual charms, either start from an existing `bundle.yaml` or define one with a list of cherry-picked charms. The following shows an example for a custom-defined `charms.yaml` (can be any file name):

```
applications:
  postgresql:
    charm: postgresql-k8s
    series: kubernetes
    channel: latest/stable
    resources:
      postgresql-image: postgres:local-image
  kafka:
    charm: kafka-k8s
```

The `charm` key is required, while the other fields will use default values if omitted. All extra keys will be ignored. The `resources` key is only necessary if a manual override of the OCI image subpath is required. See *Export OCI images* (page 30) for use cases.

Pass the `charms.yaml` to the `export charms` command like so:

```
$ store-admin export charms ./charms.yaml
Overriding postgresql-image with local registry subpath.
Downloading postgresql-k8s revision 20 (latest/stable)
  [########################################]  100%
Downloading resources for postgresql-k8s
...
Successfully exported charms to: /home/ubuntu/snap/store-admin/common/export/charms-
export-20240426T163115.tar.gz
```

The exported `tar.gz` contains the following:

```
charms-export-20240426T163115.tar.gz/
├─ bundle.yaml
├─ postgresql-k8s.tar.gz/
|  ├─ postgresql-k8s_20.charm
|  ├─ postgresql-k8s_publisher_account.assert
|  ├─ metadata.json
|  ├─ resources/
|  |  ├─ postgresql-k8s.postgresql-image_19
```

where:

- `metadata.json` is the charm metadata fetched, this is required by CHP to write the charm's channel map and other metadata to enable deployments.
- `bundle.yaml` is a copy of the user-specified export `.yaml`, provided for ease of reference.

- `postgresql-k8s_20.charm` is the binary downloaded from Charmhub, for the revision resolved.
- `postgresql-k8s_publisher_account.assert` is the account assertion of the charm publisher.
- `resources/` contain resource binaries attached to the exported charm revision.

Export snap resources

Some charms may require a specific snap revision as a resource. These charms usually run the equivalent of `snap install <snap> --revision <rev>` in their setup code (example[31]). To export snaps by revision, define a `.yaml` file of the following structure:

```
packages:
  - name: charmed-postgresql
    revision: 96
    push_channel: chp_14/edge
  - name: charmed-mysql
    revision: 97
    push_channel: 8.0/edge
```

When installing a snap by revision, the Snap Store requires that the revision exists in the snap's channel map history, i.e. the revision must have been released to any channel before it can be requested directly. Thus, `push_channel` needs to be specified to tell Snap Store Proxy the target channel for the revision. This can be a channel that exists for the snap, thereby effectively overriding the channel when the snap is pushed, or it can be an arbitrary track, which would be created in the Proxy on push.

The export `.yaml` can be supplied to the `export snaps` command like so:

```
$ store-admin export snaps --from-yaml snaps.yaml
Downloading charmed-postgresql revision 96 (chp_14/edge amd64)
  [####################################]  100%
Downloading charmed-mysql revision 97 (8.0/edge amd64)
  [####################################]  100%
Successfully exported snaps:
charmed-postgresql: /home/ubuntu/snap/store-admin/common/export/charmed-postgresql-
20240429T122503.tar.gz
charmed-mysql: /home/ubuntu/snap/store-admin/common/export/charmed-mysql-20240429T122503.
tar.gz
```

[31] https://github.com/canonical/postgresql-operator/blob/9614915048ba612bb4be6a5fd8c752a46bb051ed/lib/charms/operator_libs_linux/v2/snap.py#L460

Export OCI images

A local OCI registry needs to be set up to enable charms with OCI image resources. On charm export, the OCI image metadata blob is written to the `resources` directory, e.g. for `postgresql-k8s`:

```
{
  "ImageName": "registry.jujucharms.com/charm/kotcfrohea62xreenq1q75n1lyspke0qkurhk/
postgresql-image@sha256:8a72e1152d4a01cd9f469...",
  "Password": "MDAxOGxvY2F0aW9uIGNoYXJtc3Rvcm...",
  "Username": "docker-registry"
}
```

The image itself needs to be exported using a separate tool such as `skopeo`, which can transfer images between registries, and between a registry and a local directory, while preserving the image hash.

For example, to save the above image to a local directory:

```
$ skopeo copy docker://registry.jujucharms.com/charm/
kotcfrohea62xreenq1q75n1lyspke0qkurhk/postgresql-image@sha256:8a72e1152d4a0... --src-
creds=docker-registry:MDAxOGxvY2F0aW9... dir:/home/ubuntu/<target-dir>
```

The directory can then be manually copied to the air-gapped registry host, then pushed to the registry like so:

```
$ skopeo copy dir:/home/ubuntu/<copied-dir> docker://<local-registry-domain>/charm/
kotcfrohea62xreenq1q75n1lyspke0qkurhk/postgresql-image@sha256:8a72e1152d4a0... --dest-
creds=<local-registry-username>:<local-registry-password>
```

By default, if no override is supplied via the `resources` key in the `.yaml` supplied for charm export, Charmhub Proxy will assume an identical local registry image path (excluding the domain but including `charm/` and including the sha256 tag). When a deployment is requested, CHP will supply a regenerated blob using the local domain URL and credentials configured.

The `skopeo` commands above pushes the image to the same path in the local registry and saves the effort of manually remapping resources. If required, the image can be pushed to a custom path, but a mapping must be defined for the resource as in the example `charms.yaml` in *Export charms* (page 28).

Import Packages

Once the exported charm tar file is on the on-prem store host, they should be moved to the `/var/snap/snap-store-proxy/common/charms-to-push/` directory, from where they can be imported.

Example of importing `charms-export-20240429T090849.tar.gz` from the previous example:

```
sudo snap-store-proxy push-charms /var/snap/snap-store-proxy/common/charms-to-push/charms-
export-20240429T090849.tar.gz
```

Example of importing a `cos-lite` bundle:

```
sudo snap-store-proxy push-charm-bundle /var/snap/snap-store-proxy/common/charms-to-push/
cos-lite-20240401T172030.tar.gz
```

When re-importing charms or importing other revisions, make sure to provide the `--push-channel-map`.

After importing, the charms/bundles are then available to be managed with Juju commands.

- When importing machine charms that depend on a snap for functionality, you must first manually *import the required snap* (page 15).
- When importing Kubernetes charms, ensure that the corresponding OCI image is copied to the local registry, maintaining its original path.

Configure Juju

Ensure you have Juju configured along with the necessary cloud environment.

For production environments, particularly if you need self-signed TLS certificates to function correctly, set up the Juju controller on a non-Kubernetes instance. Even if the Juju controller isn't hosted on a Kubernetes cluster, it can still manage Kubernetes models. This flexibility allows the controller to operate from a different environment, such as a virtual machine or an LXD container, while still effectively orchestrating resources within Kubernetes.

This setup allows you to incorporate a self-signed certificate within the `cloudinit` configuration, enabling the Juju controller to trust the certificate.

First, you need to prepare the Juju configuration file. In this file, override the default URLs for Charmhub and Snap Store. Additionally, if you're using a self-signed certificate for Charmhub, include it in the trusted certificates section of `cloudinit-userdata`.

Self-signed certificate

You can create a self signed certificate for the Snap store proxy with the following command:

```
sudo snap-store-proxy import-certificate --selfsigned
```

After it's created, you can retrieve the public key from the configuration:

```
snap-store-proxy config proxy.tls.cert | cat > tls-cert.crt
```

When using a self-signed certificate, it's crucial to ensure that the underlying operating system where the Juju client is running trusts the certificate. You can achieve this by adding the certificate to the system's trusted store. You can achieve that with the following commands:

```
sudo cp your_certificate.crt /usr/local/share/ca-certificates/
sudo update-ca-certificates
```

Configuration file

Example of a Juju configuration `.yaml`. Note that ca-certs list is necessary only when using self-signed certificate for the local Charmhub.

```
cloudinit-userdata: |
  ca-certs:
    trusted:
    - |
      -----BEGIN CERTIFICATE-----
      MIIFGjCCAwKgAwIBAgIUQzfvzTdygyQdNx69x/sMzu/xWF4wDQYJKoZIhvcNAQEL
      ...
      mfV/T8n8J15gfYTAyKs=
      -----END CERTIFICATE-----
charmhub-url: https://local-charmhub.internal
snap-store-proxy-url: https://local-charmhub.internal
```

Store this file in a Juju accessible path e.g. `/var/snap/juju/common/juju-config.yaml`.

Controller and Model setup

After configuring the certificate, the next steps depend on your deployment target. If you plan to deploy to a Kubernetes (k8s) cloud, you'll need to add the substrate to this controller to facilitate the deployment. However, if you're deploying machine charms, this additional step is not necessary.

```
juju bootstrap lxd machine-controller --config=/var/snap/juju/common/juju-config.yaml
```

The example below assumes that an LXD cloud is already set up and utilises it to create a Juju controller:

```
juju add-k8s k8s-cloud --controller=machine-controller
```

We can then create a model using the following example:

```
juju add-model test-model k8s-cloud --config /var/snap/juju/common/juju-config.yaml
```

In case we want to deploy to a non k8s cloud, we can skip the cloud parameter:

```
juju add-model test-model --config /var/snap/juju/common/juju-config.yaml
```

Deploy

Finally, we can deploy the charms using the standard Juju command.

```
juju deploy cos-lite
```

All charm management commands associated with the controller, including `refresh` and `remove`, work seamlessly out of the box.

Info commands

Some Juju commands are not tied to a controller and instead require the setup of an environment variable. Notably, this includes the `info` and `download` commands. In such cases, you can set the `CHARMHUB_URL` environment variable before executing these commands.

```
CHARMHUB_URL="https://local-charmhub.internal" juju info cos-lite
```

To make this change more permanent, you can add the variable to the `.bashrc` file in the user's home folder.

This ensures that the custom URL is consistently used whenever Juju commands[32] are run from the terminal.

1.1.9. Troubleshooting

To check egress firewall rules:

```
snap-proxy check-connections
```

Logs are available in systemd logs:

```
snap logs snap-store-proxy
```

or:

```
journalctl -u 'snap.snap-store-proxy.*'
```

The snap-proxy snap includes multiple systemd services, the status of which can be checked with:

```
snap-proxy status
```

Or:

```
sudo systemctl status -a 'snap.snap-store-proxy.*'
```

To restart the snap-proxy services, run:

```
sudo snap restart snap-proxy
```

The download cache is at `/var/snap/snap-store-proxy/current/nginx/cache`. The default limit is 2GB, this can be changed with:

```
sudo snap-proxy config proxy.cache.size=4096  # in mb
```

[32] https://juju.is/docs/juju/manage-charms-or-bundles

Moving to a new hostname

If you need to move the snap-proxy to a new hostname, you can do:

```
sudo snap-proxy config proxy.domain=NEWDOMAIN
sudo snap-proxy reregister
```

This perform another registration cycle and update the assertion file with the new domain name. Then you will need to run `snap ack` on the client devices to replace the existing assertion.

Documentation

This documentation is shipped with the snap, and available at:

```
http://MY-PROXY/docs/
```

Bug reporting

Please file bugs against this project on Launchpad[33]

Known issues

1. The `snap download` command doesn't do the download of the snap through `snapd` service, and therefore doesn't know about the Snap Store Proxy and will try to fetch the snap directly. Forum thread[34]
2. Need to be root when configuring the snap proxy. Forum thread[35]

1.2. Snap Store Proxy Reference

Our *Reference* section contains technical details (such as the Overrides API specs and authentication mechanism), and other supplementary reference materials.

Reference	How the Proxy works
Overrides API (page 35)	API specs for the Overrides API
API authentication (page 38)	API authentication for Proxy admins when using the Overrides API
Feature list (page 41)	A summarised list of features of the Proxy
Product whitepaper (page 41)	Whitepaper on managing snaps in an enterprise environment with the Proxy
Cryptography (page 41)	An Outline of the usage of cryptographic technology

Alternatively, our *Tutorials* section contain step-by-step tutorials to help outline what the Proxy is capable of while helping you achieve specific aims.

[33] https://bugs.launchpad.net/snapstore
[34] https://forum.snapcraft.io/t/improvements-in-snap-download/1422
[35] https://forum.snapcraft.io/t/should-snapctl-set-in-apps-trigger-the-configure-hook/2032/7

If you have a specific goal, but are already familiar with the Snap Store Proxy, 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.

Finally, for a better understanding of how the Snap Store Proxy works, our *Explanation* section enables you to expand your knowledge.

1.2.1. Overrides API

The Overrides API supports two operations: list overrides, and set overrides.

All operations require an "Authorisation" header, as described in *API authentication* (page 38).

List overrides

To list overrides for a snap name, perform a `GET` request to `/v2/metadata/overrides/{snap_name}`:

Example: to list overrides for `snap_a`:

Request:

```
GET /v2/metadata/overrides/snap_a HTTP/1.1
Host: <store domain>
Accept: application/json
X-Ubuntu-Series: 16
```

Note the `X-Ubuntu-Series` header.

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
...

{
    "overrides": [
        {
            "snap_id": ...,
            "snap_name": "snap_a",
            "revision": 20,
            "upstream_revision": 23,
            "channel": "stable",
            "architecture": "x86",
            "series": "16"
        }
    ]
}
```

The JSON Schema for the response document is:

```
{
    "type": "object",
    "properties": {
        "overrides": {
            "type": "array",
```

(continues on next page)

(continued from previous page)

```
            "items": {
                "type": "object",
                "properties": {
                    "snap_id": {"type": "string"},
                    "snap_name": {"type": "string"},
                    "revision": {
                        "type": "integer",
                        "minimum": 1,
                    },
                    "upstream_revision": {
                        "type": ["integer", "null"],
                        "minimum": 1,
                    },
                    "channel": {"type": "string"},
                    "architecture": {"type": "string"},
                    "series": {"type": "string"},
                },
                "required": [
                    "snap_id", "snap_name", "revision", "upstream_revision",
                    "channel", "architecture", "series",
                ],
                "additionalProperties": False,
            },
        },
    },
    "required": ["overrides"],
    "additionalProperties": False,
}
```

Set overrides

To set overrides, perform a `POST` request to `/v2/metadata/overrides`.

Example: override `snap_a` to revision 20 on the stable channel, and remove any overrides for `snap_b` on the candidate channel.

Request:

```
POST /v2/metadata/overrides HTTP/1.1
Host: <store domain>
Accept: application/json
Content-Type: application/json

[
    {
        "snap_name": "snap_a",
        "revision": 20,
        "channel": "stable",
        "series": "16"
    },
    {
        "snap_name": "snap_b",
        "revision": null,
        "channel": "candidate",
        "series": "16"
```

(continues on next page)

(continued from previous page)

```
    }
]
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
...

{
    "overrides": [
        {
            "snap_id": ...,
            "snap_name": "snap_a",
            "revision": 20,
            "upstream_revision": 23,
            "channel": "stable",
            "architecture": "x86",
            "series": "16"
        }, {
            "snap_id": ...,
            "snap_name": "snap_b",
            "revision": null,
            "upstream_revision": 13,
            "channel": "candidate",
            "architecture": "x86",
            "series": "16"
        }
    ]
}
```

The request body format should match the following JSON schema.

```
{
    "type": "array",
    "items": {
        "type": "object",
        "properties": {
            "snap_name": {"type": "string"},
            "revision": {
                "type": ["integer", "null"],
                "minimum": 1,
            },
            "channel": {"type": "string"},
            "series": {"type": "string"},
        },
        "required": ["snap_name", "revision", "channel", "series"],
        "additionalProperties": False,
    },
    "minItems": 1,
}
```

To delete an override, simply set its revision field to `null`. The “series” field must be set to “16” currently.

The response includes the results of the operation. The response format is the same as for

listing overrides, except that “revision” can be `null` if an override was deleted.

1.2.2. API authentication

Authentication with the API requires a valid Ubuntu SSO user. This user must be configured as an admin using the `snap-proxy` tool:

```
snap-proxy add-admin user@example.com
```

The Snap Store and Snap Store Proxy use macaroons for authentication, which are a kind of bearer token that can be constrained and that can be authorised by third-party services. We strongly recommend using pymacaroons[36] or libmacaroons[37] to work with these tokens.

If you want to understand more about how macaroons work, refer to the original paper[38].

To login, you must first get a root macaroon from the snap store proxy, then discharge (verify) that macaroon with Ubuntu SSO[39].

Root Macaroon

To get a root macaroon:

Request:

```
POST /v2/auth/issue-store-admin HTTP/1.1
Host: <store domain>
Accept: application/json
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
...

{
    "macaroon": "..."
}
```

The response is simple and matches this JSON Schema:

```
{
    'type': 'object',
    'properties': {
        'macaroon': {'type': 'string'},
    },
    'required': ['macaroon'],
    'additionalProperties': False,
}
```

This is your main authentication token, and should be stored persistently.

[36] https://github.com/ecordell/pymacaroons
[37] https://github.com/rescrv/libmacaroons
[38] https://research.google.com/pubs/pub41892.html
[39] https://login.ubuntu.com/

Discharge Ubuntu SSO caveat

The root macaroon contains a caveat that the user must have a valid Ubuntu SSO account. To prove that is the case, we need to discharge that caveat with Ubuntu SSO.

You need to deserialise this root macaroon, and extract the caveat ID with the location `login.ubuntu.com`. For example, using pymacaroons:

```
macaroon = pymacaroons.Macaroon.deserialize(root_macaroon)
for caveat in macaroon.caveats:
    if caveat.location == 'login.ubuntu.com':
        return caveat.caveat_id
```

Then we need to discharge the caveat with Ubuntu SSO.

Request:

```
POST /api/v2/tokens/discharge HTTP/1.1
Host: login.ubuntu.com
Accept: application/json
Content-Type: application/json

{
    "email": ...,
    "password": ...,
    "otp": ...,  # if user account has 2FA enabled
    "caveat_id": ...
}
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
...

{
    "discharge_macaroon": "<discharge macaroon>",
}
```

Note

For more detailed responses from Ubuntu SSO, particularly handling invalid credentials and 2FA, see the general Ubuntu SSO documentation for OAuth tokens[40], which is also used by the macaroon discharge endpoint.

You will need to persist the raw root macaroon and the raw discharge macaroon. Together, these are your authentication.

[40] http://canonical-identity-provider.readthedocs.io/en/latest/resources/token.html

Request authentication

To authenticate a request, you must bind the discharge macaroon to the root macaroon, and send that as your value in an 'Authorisation' HTTP header.

For example, with pymacaroons:

```
root = pymacaroon.Macaroon.deserialize(root_raw)
discharge = pymacaroons.Macaroon.deserialize(discharge_raw)
bound = root.prepare_for_request(discharge)
header = 'Macaroon root="{}", discharge="{}"'.format(root_raw, bound.serialize())
```

Note

If your discharge macaroon has expired, it will be indicated by indicated by a 401 status code, and a header: `HTTP/1.1 401 Unauthorized WWW-Authenticate: Macaroon needs_refresh=1`

In this case you will need to refresh your discharge macaroon, described below, and retry the request.

Refreshing the discharge macaroon

Your discharge macaroon has an expiry, and needs refreshing with Ubuntu SSO periodically.

To do so, simply:

Request:

```
POST /api/v2/tokens/refresh HTTP/1.1
Host: login.ubuntu.com
Accept: application/json
Content-Type: application/json

{
    "discharge_macaroon": "<discharge>"
}
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
...

{
    "discharge_macaroon": "<new discharge>",
}
```

Update and store persistently this new discharge macaroon for later use.

1.2.3. Feature list

- Network control
  - Provides a means to access the Snap Store for devices with restricted network access
  - Snap Store Proxy can communicate with the Snap Store directly or through an HTTPS forward proxy
- Caching of the downloaded snaps
- *Overriding revisions* (page 10) of specific snaps for all connected devices
- Management options
  - `snap-proxy` CLI interface included with the Snap Store Proxy[41] snap
  - Remote management using the Snap Store Proxy Client[42] or a *RESTful API* (page 35).
- *Offline mode* (page 12).

Note

Unless it is deliberately set up as *offline* (page 12), a Snap Store Proxy needs to be online and connected to the general Snap Store[43]. This is a requirement, even though Snap Store Proxy caches downloaded snap files, which substantially reduces internet traffic. There's currently no generally available offline mode for the Snap Store Proxy itself. See *Network Connectivity* (page 6) for the `snap-proxy check-connections` command and the up-to-date Network requirements for Snappy[44] post for a list of domains Snap Store Proxy needs access to.

1.2.4. Whitepaper

Learn more about how the Snap Store Proxy overcomes challenges presented by restricted networks and management policies from this whitepaper on Enterprise Snap Management[45].

1.2.5. Cryptography

Various Cryptographic technologies are used to enable secure Snap Store Proxy operation. Below are the functionalities of the Snap Store Proxy that use cryptographic technologies, and the details of the cryptographic technologies used.

- **Signing assertions**: the Snap Store Proxy signs various assertions[46]. The key ID of the signing key is encoded with SHA3-384, and the assertion is signed with RSA.
- **Hash of artefacts**: the Snap Store Proxy generates many hashes of an uploaded artefact using SHA3-384, SHA256 and SHA512 to ensure the uniqueness and integrity of the artefact.

[41] https://snapcraft.io/snap-store-proxy
[42] https://snapcraft.io/snap-store-proxy-client
[43] https://snapcraft.io/store
[44] https://forum.snapcraft.io/t/network-requirements-for-snappy/5147
[45] https://ubuntu.com/engage/enterprise-snap-management
[46] https://ubuntu.com/core/docs/reference/assertions

- **OCI charm resources credentials**: an OCI runtime (e.g. microk8s[47]) must authenticate against the Snap Store Proxy in order to download the OCI charm resources[48]. The credentials are encoded as JWT that are signed with RSA.
- **Signing nonce**: A nonce is used as additional security for REST API access. RSA is used to sign and verify the nonce.

Function	Exposed	Technology	Package/Library
Signing assertions	Yes	SHA3-384, RSA 4096/8192	snapd[49], lp-signing[50]
Hash of artefacts	Yes	SHA3-384, SHA256, SHA512	review-tools[51]
OCI charm resources credentials	Yes	RSA 4096, JWT	cryptography[52], pyjwt[53], py-macaroon-bakery[54]
Signing nonce	Yes	RSA 4096	cryptography[55], pem[56]

[47] https://microk8s.io/docs
[48] https://juju.is/docs/juju/charm-resource
[49] https://github.com/canonical/snapd
[50] https://launchpad.net/lp-signing
[51] https://launchpad.net/review-tools
[52] https://github.com/pyca/cryptography
[53] https://github.com/jpadilla/pyjwt
[54] https://github.com/go-macaroon-bakery/py-macaroon-bakery
[55] https://github.com/pyca/cryptography
[56] https://github.com/hynek/pem