LXD image builder

Canonical Group Ltd

Sep 11, 2024

CONTENTS

`lxd-imagebuilder` is an image building tool for LXC and LXD.

Its modern design uses pre-built official images whenever available and supports a variety of modifications on the base image. `lxd-imagebuilder` creates LXC or LXD images, or just a plain root file system, from a declarative image definition (in YAML format) that defines the source of the image, its package manager, what packages to install or remove for specific image variants, OS releases and architectures, as well as additional files to generate and arbitrary actions to execute as part of the image build process.

`lxd-imagebuilder` can be used to create custom images that can be used as the base for LXC containers or LXD instances.

`lxd-imagebuilder` is used to build the images on the LXD image server. You can also use it to build images from ISO files that require licenses and therefore cannot be distributed.

CHAPTER

ONE

IN THIS DOCUMENTATION

Tutorial

Start here: a hands-on introduction to `lxd-imagebuilder` and `simplestream-maintainer` for new users

How-to guides

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

Reference

Technical information - specifications, APIs, architecture for `lxd-imagebuilder` and `simplestream-maintainer`

Explanation (coming)

Discussion and clarification of key topics

CHAPTER

TWO

PROJECT AND COMMUNITY

`lxd-imagebuilder` is free software and released under AGPLv3. It's an open source project that warmly welcomes community projects, contributions, suggestions, fixes and constructive feedback.

- Contribute to the project
- Discuss on IRC (see Getting started with IRC if needed)
- Ask and answer questions on the forum

2.1 Tutorials

These tutorials guide you through the usage of `lxd-imagebuilder` and `simplestream-maintainer`.

2.1.1 Use `lxd-imagebuilder` to create images

This guide shows you how to create an image for LXD or LXC.

Before you start, you must install `lxd-imagebuilder`. See *How to install `lxd-imagebuilder` and `simplestream-maintainer`* for instructions.

Create an image

To create an image, first create a directory where you will be placing the images, and enter that directory.

```
mkdir -p $HOME/Images/ubuntu/
cd $HOME/Images/ubuntu/
```

Then, copy one of the example YAML configuration files for images into this directory.

> **Note**
>
> The YAML configuration file contains an image template that gives instructions to LXD imagebuilder.
>
> LXD imagebuilder provides examples of YAML files for various distributions in the examples directory. `scheme.yaml` is a standard template that includes all available options.
>
> Official LXD templates for various distributions are available in the `lxd-ci` repository.

In this example, we are creating an Ubuntu image.

```
cp $HOME/go/src/github.com/canonical/lxd-imagebuilder/doc/examples/ubuntu.yaml ubuntu.
↪yaml
```

Edit the template file

Optionally, you can do some edits to the YAML configuration file. You can define the following keys:

Section	Description	Documentation
`image`	Defines distribution, architecture, release etc.	*Image*
`source`	Defines main package source, keys etc.	*Source*
`targets`	Defines configuration for specific targets (e.g. LXD, instances etc.)	*Targets*
`files`	Defines generators to modify files	*Generators*
`packages`	Defines packages for install or removal; adds repositories	*Package management*
`actions`	Defines scripts to be run after specific steps during image building	*Actions*
`mappings`	Maps different terms for architectures for specific distributions (e.g. `x86_64: amd64`)	*Mappings*

Tip

When building a VM image, you should either build an image with cloud-init support (provides automatic size growth) or set a higher size in the template, because the standard size is relatively small (10 GiB). Alternatively, you can also grow it manually.

Build and launch the image

The steps for building and launching the image depend on whether you want to use it with LXD or with LXC.

Create an image for LXD

To build an image for LXD, run `lxd-imagebuilder`. We are using the `build-lxd` option to create an image for LXD.

- To create a container image:

```
sudo $HOME/go/bin/lxd-imagebuilder build-lxd ubuntu.yaml
```

- To create a VM image:

```
sudo $HOME/go/bin/lxd-imagebuilder build-lxd ubuntu.yaml --vm
```

See *LXD image* for more information about the `build-lxd` command.

If the command is successful, you will get an output similar to the following (for a container image). The `lxd.tar.xz` file is the description of the container image. The `rootfs.squasfs` file is the root file system (rootfs) of the container image. The set of these two files is the *container image*.

```
$ ls -l
total 100960
-rw-r--r-- 1 root   root         676 Oct  3 16:15 lxd.tar.xz
-rw-r--r-- 1 root   root   103370752 Oct  3 16:15 rootfs.squashfs
-rw-r--r-- 1 ubuntu ubuntu      7449 Oct  3 16:03 ubuntu.yaml
$
```

Add the image to LXD

To add the image to an LXD installation, use the `lxc image import` command as follows.

```
$ lxc image import lxd.tar.xz rootfs.squashfs --alias mycontainerimage
Image imported with fingerprint: 009349195858651a0f883de804e64eb82e0ac8c0bc51880
```

See How to copy and import images for detailed information.

Let's look at the image in LXD. The `ubuntu.yaml` had a setting to create an Ubuntu 20.04 (`focal`) image. The size is 98.58MB.

```
$ lxc image list mycontainerimage
+------------------+--------------+--------+--------------+--------+---------+-----------
↪------------------+
|      ALIAS       | FINGERPRINT  | PUBLIC | DESCRIPTION  |  ARCH  |  SIZE   |        ␣
↪UPLOAD DATE         |
+------------------+--------------+--------+--------------+--------+---------+-----------
↪------------------+
| mycontainerimage | 009349195858 | no     | Ubuntu focal | x86_64 | 98.58MB | Oct 3,␣
↪2020 at 5:10pm (UTC) |
+------------------+--------------+--------+--------------+--------+---------+-----------
↪------------------+
```

Launch an LXD container from the container image

To launch a container from the freshly created image, use `lxc launch` as follows. Note that you do not specify a repository for the image (like `ubuntu:` or `images:`) because the image is located locally.

```
$ lxc launch mycontainerimage c1
Creating c1
Starting c1
```

Create an image for LXC

Using LXC containers instead of LXD may require the installation of `lxc-utils`. Having both LXC and LXD installed on the same system will probably cause confusion. Use of raw LXC is generally discouraged due to the lack of automatic AppArmor protection.

To create an image for LXC, use the following command:

```
$ sudo $HOME/go/bin/lxd-imagebuilder build-lxc ubuntu.yaml
$ ls -l
```

(continues on next page)

(continued from previous page)

```
total 87340
-rw-r--r-- 1 root root      740 Jan 19 03:15 meta.tar.xz
-rw-r--r-- 1 root root 89421136 Jan 19 03:15 rootfs.tar.xz
-rw-r--r-- 1 root root     4798 Jan 19 02:42 ubuntu.yaml
```

See *LXC image* for more information about the `build-lxc` command.

Add the container image to LXC

To add the container image to a LXC installation, use the `lxc-create` command as follows.

```
lxc-create -n myContainerImage -t local -- --metadata meta.tar.xz --fstree rootfs.tar.xz
```

Launch a LXC container from the container image

Then start the container with

```
lxc-start -n myContainerImage
```

Repack Windows ISO

With LXD it's possible to run Windows VMs. All you need is a Windows ISO and a bunch of drivers. To make the installation a bit easier, `lxd-imagebuilder` added the `repack-windows` command. It takes a Windows ISO, and repacks it together with the necessary drivers.

The `lxd-imagebuilder` will automatically detect the Windows version, however, it is possible to set the version manually using `--windows-version` flag, which allows the following values:

Flag value	Version
`w11`	Windows 11
`w10`	Windows 10
`w8`	Windows 8
`w8.1`	Windows 8.1
`w7`	Windows 7
`xp`	Windows XP
`2k22`	Windows Server 2022
`2k19`	Windows Server 2019
`2k16`	Windows Server 2016
`2k12`	Windows Server 2012
`2k12r2`	Windows Server 2012 R2
`2k8`	Windows Server 2008
`2k8r2`	Windows Server 2008 R2
`2k3`	Windows Server 2003

When repacking a Windows ISO, `lxd-imagebuilder` uses external tools that may need to be installed. On a Ubuntu/Debian system, those can be installed with:

```
sudo apt-get install -y --no-install-recommends genisoimage libwin-hivex-perl rsync␣
↪wimtools
```

Here's how to repack a Windows ISO:

```
sudo lxd-imagebuilder repack-windows path/to/Windows.iso path/to/Windows-repacked.iso
```

More information on `repack-windows` can be found by running

```
lxd-imagebuilder repack-windows -h
```

Install Windows

Run the following commands to initialize the VM, add a TPM device, increase the allocated disk space, CPU, memory and finally attach the full path of your prepared ISO file.

```
lxc init win11 --empty --vm
lxc config device add win11 vtpm tpm path=/dev/tpm0
lxc config device override win11 root size=50GiB
lxc config set win11 limits.cpu=4 limits.memory=8GiB
lxc config device add win11 iso disk source=/path/to/Windows-repacked.iso boot.
↪priority=10
```

Now, the VM win11 has been configured and it is ready to be started. The following command starts the virtual machine and opens up a VGA console so that we go through the graphical installation of Windows.

```
lxc start win11 --console=vga
```

Once done with the manual installation process, the ISO can be removed to speed up next boots by avoiding the prompt to boot from it.

```
lxc config device remove win11 iso
```

2.1.2 Use `simplestream-maintainer` to automate image server maintenance

`simplestream-maintainer` is capable of generating a simple streams product catalog and removing expired or invalid product versions. However, it is a simple CLI tool that has to be invoked every time an action needs to be done.

To automate the process of maintaining a simple streams image server, we recommend triggering the build and prune commands periodically, either via cron jobs or systemd units.

On servers that host hundreds or more images, the build process can take quite a long time because it has to calculate missing hashes and generate missing delta files. In such cases, we recommend using systemd units to prevent triggering unnecessary builds if the previous build has not finished yet.

Example using systemd units

First create a systemd service file `/etc/systemd/system/simplestream-maintainer.service` which runs `simplestream-maintainer build` and `prune` commands as in the provided example below. Ensure `<simplestream_dir>` is replaced with an actual simple streams root directory and `<simplestream_user>` with a user that has write permissions to that directory.

```
# /etc/systemd/system/simplestream-maintainer.service
[Unit]
Description=Simplestream maintainer
```

(continues on next page)

(continued from previous page)

```
ConditionPathIsDirectory="<simplestream_dir>/images"

[Service]
Type=oneshot
User="<simplestream_user>"
Environment=TZ=UTC

# Commands are executed in the exact same order as specified.
ExecStart=simplestream-maintainer build "<simplestream_dir>" --logformat json --loglevel␣
↪warn --workers 4
ExecStart=simplestream-maintainer prune "<simplestream_dir>" --logformat json --loglevel␣
↪warn --retain-builds 3 --dangling

# Processes running at "idle" level get CPU time only when no one else needs it.
# This prevents simplestream-maintainer from consuming the computational resources when
# they are used to serve the images.
CPUSchedulingPolicy=idle

# Processes running at "idle" level get I/O time only when no one else needs the disk.
# This prevents simplestream-maintainer from consuming the disk I/O when it is used to
# serve the images.
IOSchedulingClass=idle
```

To start the systemd service periodically, create a new systemd timer file `/etc/systemd/system/simplestream-maintainer.timer`. The following example triggers the previously created systemd service each hour with a random 5 minute offset.

```
# /etc/systemd/system/simplestream-maintainer.timer
[Unit]
Description=Simplestream maintainer timer

[Timer]
OnCalendar=hourly
RandomizedDelaySec=5m
Persistent=true

[Install]
WantedBy=timers.target
```

2.2 How-to guides

These how-to guides cover key operations and processes in `lxd-imagebuilder`.

2.2.1 How to install `lxd-imagebuilder` and `simplestream-maintainer`

Installing from package

The `lxd-imagebuilder` and `simplestream-maintainer` tools are available in the `lxd-imagebuilder` snap from the Snap Store.

```
sudo snap install lxd-imagebuilder --classic
```

Installing from source

To compile from source, first install the Go programming language, and some other dependencies.

- Debian-based:

```
sudo apt update
sudo apt install -y golang-go debootstrap rsync gpg squashfs-tools git make xdelta3
```

- ArchLinux-based:

```
sudo pacman -Syu
sudo pacman -S go debootstrap rsync gnupg squashfs-tools git make xdelta3 --needed
```

NOTE: Go 1.22 or higher is required. If your package manager doesn't provide a recent enough version, get it from upstream.

Second, download the source code of the `lxd-imagebuilder` repository (this repository).

```
mkdir -p $HOME/go/src/github.com/canonical/
cd $HOME/go/src/github.com/canonical/
git clone https://github.com/canonical/lxd-imagebuilder
```

Third, enter the directory with the source code of `lxd-imagebuilder` and run `make` to compile the source code. This will generate the executable programs `lxd-imagebuilder` and `simplestream-maintainer` in `$HOME/go/bin`.

```
cd ./lxd-imagebuilder
make
```

You may also add the directory `$HOME/go/bin/` to your $PATH so that you do not need to run the command with the full path.

2.2.2 How to build images

Plain rootfs

```
$ lxd-imagebuilder build-dir --help
Build plain rootfs

Usage:
  lxd-imagebuilder build-dir <filename|-> <target dir> [flags]

Flags:
```

(continues on next page)

(continued from previous page)

```
  -h, --help              help for build-dir
      --keep-sources      Keep sources after build (default true)
      --sources-dir       Sources directory for distribution tarballs (default "/tmp/lxd-
↪imagebuilder")
      --with-post-files   Run post-files actions

Global Flags:
      --cache-dir         Cache directory
      --cleanup           Clean up cache directory (default true)
      --debug             Enable debug output
      --disable-overlay   Disable the use of filesystem overlays
  -o, --options           Override options (list of key=value)
  -t, --timeout           Timeout in seconds
      --version           Print version number
```

To build a plain rootfs, run `lxd-imagebuilder build-dir`. The command takes an image definition file and an output directory as positional arguments. Running `build-dir` is useful if one wants to build both LXC and LXD images. In that case one can simply run

```
lxd-imagebuilder build-dir def.yaml /path/to/rootfs
lxd-imagebuilder pack-lxc def.yaml /path/to/rootfs /path/to/output
lxd-imagebuilder pack-lxd def.yaml /path/to/rootfs /path/to/output
```

LXC image

```
$ lxd-imagebuilder build-lxc --help
Build LXC image from scratch

The compression can be set with the --compression flag. I can take one of the
following values:
  - bzip2
  - gzip
  - lzip
  - lzma
  - lzo
  - lzop
  - xz (default)
  - zstd

For supported compression methods, a compression level can be specified with
method-N, where N is an integer, e.g. gzip-9.

Usage:
  lxd-imagebuilder build-lxc <filename|-> [target dir] [--compression=COMPRESSION]␣
↪[flags]

Flags:
      --compression    Type of compression to use (default "xz")
  -h, --help           help for build-lxc
      --keep-sources   Keep sources after build (default true)
      --sources-dir    Sources directory for distribution tarballs (default "/tmp/lxd-
```

(continues on next page)

(continued from previous page)

```
↪imagebuilder")

Global Flags:
      --cache-dir         Cache directory
      --cleanup           Clean up cache directory (default true)
      --debug             Enable debug output
      --disable-overlay   Disable the use of filesystem overlays
  -o, --options           Override options (list of key=value)
  -t, --timeout           Timeout in seconds
      --version           Print version number
```

Running the `build-lxc` sub-command creates a LXC image. It outputs two files `rootfs.tar.xz` and `meta.tar.xz`. After building the image, the rootfs will be destroyed.

The `pack-lxc` sub-command can be used to create an image from an existing rootfs. The rootfs won't be deleted afterwards.

LXD image

```
$ lxd-imagebuilder build-lxd --help
Build LXD image from scratch

Depending on the type, it either outputs a unified (single tarball)
or split image (tarball + squashfs or qcow2 image). The --type flag can take one of the
following values:
  - split (default)
  - unified


The compression can be set with the --compression flag. I can take one of the
following values:
  - bzip2
  - gzip
  - lzip
  - lzma
  - lzo
  - lzop
  - xz (default)
  - zstd

For supported compression methods, a compression level can be specified with
method-N, where N is an integer, e.g. gzip-9.

Usage:
  lxd-imagebuilder build-lxd <filename|-> [target dir] [--type=TYPE] [--
↪compression=COMPRESSION] [--import-into-lxd] [flags]

Flags:
      --compression               Type of compression to use (default "xz")
  -h, --help                      help for build-lxd
      --import-into-lxd[="-"]     Import built image into LXD
```

(continues on next page)

(continued from previous page)

```
      --keep-sources              Keep sources after build (default true)
      --sources-dir               Sources directory for distribution tarballs (default "/
↪tmp/lxd-imagebuilder")
      --type                      Type of tarball to create (default "split")
      --vm                        Create a qcow2 image for VMs

Global Flags:
      --cache-dir         Cache directory
      --cleanup           Clean up cache directory (default true)
      --debug             Enable debug output
      --disable-overlay   Disable the use of filesystem overlays
  -o, --options           Override options (list of key=value)
  -t, --timeout           Timeout in seconds
      --version           Print version number
```

Running the `build-lxd` sub-command creates an LXD image. If `--type=split`, it outputs two files. The metadata tarball will always be named `lxd.tar.xz`. When creating a container image, the second file will be `rootfs.squashfs`. When creating a VM image, the second file will be `disk.qcow2`. If `--type=unified`, a unified tarball named `<image.name>.tar.xz` is created. See the *image section* for more on the image name.

If `--compression` is set, the tarballs will use the provided compression instead of `xz`.

Setting `--vm` will create a `qcow2` image which is used for virtual machines.

If `--import-into-lxd` is set, the resulting image is imported into LXD. It basically runs `lxc image import <image>`. Per default, it doesn't create an alias. This can be changed by calling it as `--import-into-lxd=<alias>`.

After building the image, the rootfs will be destroyed.

The `pack-lxd` sub-command can be used to create an image from an existing rootfs. The rootfs won't be deleted afterwards.

2.2.3 How to build the simple streams index

```
Usage:
  simplestream-maintainer build <path> [flags]

Flags:
      --build-webpage           Build index.html
  -d, --image-dir strings       Image directory (relative to path argument) (default␣
↪[images])
      --stream-version string   Stream version (default "v1")
      --workers int             Maximum number of concurrent operations (default "<max_
↪cpu>/2")
```

The build command is used to update the product catalog and generate a corresponding simple streams index file. This is achieved by first reading the existing product catalog, and then traversing through the actual directory tree of the stream to detect the differences.

Each new product version is analyzed to ensure it is complete, which means the version contains all the required files (metadata and rootfs) and is not hidden. For complete versions, the file hashes are calculated and, if necessary, delta files are generated.

The final product catalog is generated in `streams/<stream_version>/<stream>.json` and the index file in `streams/<stream_version>/index.json`.

Checksum verification

If a specific version contains a `SHA256SUMS` file, checksums are parsed from it, and compared against the calculated file hashes. If there is a mismatch, the version is not included in the final product catalog.

This allows verification of images that are built on the remote location and pushed to the simple streams server.

Webpage

The build command allows to optionally generate a static webpage (`index.html`) in the stream's root directory. The resulting webpage contains a table of all products that are extracted from the final product catalog.

2.2.4 How to prune images hosted on the simple streams server

```
Usage:
  simplestream-maintainer prune <path> [flags]

Flags:
      --dangling                Remove dangling product versions (not referenced from a␣
↪product catalog)
  -d, --image-dir strings       Image directory (relative to path argument) (default␣
↪[images])
      --retain-builds int       Maximum number of product versions to retain (default 10)
      --retain-days int         Maximum number of days to retain any product version
      --stream-version string   Stream version (default "v1")
```

The prune command is used to remove no longer needed product versions (images). Once pruning is complete, the product catalog and the simple streams index are updated accordingly.

Retention policy

Product versions are retrieved from the existing product catalog and removed according to the set retention policy.

The `--retain-builds` flag instructs `simplestream-maintainer` to keep the latest *n* versions (sorted alphabetically) and remove everything else.

The `--retain-days` flag sets the maximum age of the product version and ensures that no product version older than the specified number of days remains on the system or product catalog. By default, this flag is set to `0` which means the product versions are not pruned by age.

Dangling images

When pruning product versions, the stream's contents are retrieved from the product catalog. This means there might exist an invalid product version that was not included in the product catalog (for example, due to a missing metadata file or mismatched checksums). By default, such versions are not removed.

The `--dangling` flag instructs `simplestream-maintainer` to remove product versions that are not referenced by the product catalog. To ensure freshly uploaded or generated product versions are not accidentally removed, unreferenced product versions are removed only if they are older than 6 hours.

2.2.5 How to troubleshoot `lxd-imagebuilder`

This section covers some of the most commonly encountered problems and gives instructions for resolving them.

Cannot install into target

> Error `Cannot install into target '/var/cache/lxd-imagebuilder.123456789/rootfs' mounted with noexec or nodev`

You have installed `lxd-imagebuilder` into an LXD container and you are trying to run it. `lxd-imagebuilder` does not run in an LXD container. Run `lxd-imagebuilder` on the host, or in a VM.

Classic confinement

> Error `error: This revision of snap "lxd-imagebuilder" was published using classic confinement`

You are trying to install the `lxd-imagebuilder` snap package. The `lxd-imagebuilder` snap package has been configured to use the `classic` confinement. Therefore, when you install it, you have to add the flag `--classic` as shown above in the instructions.

Must be root

> Error `You must be root to run this tool`

You must be *root* in order to run the `lxd-imagebuilder` tool. The tool runs commands such as `mknod` that require administrative privileges. Use `sudo` when running `lxd-imagebuilder`.

2.3 Reference

The reference material in this section provides technical descriptions of `lxd-imagebuilder`.

2.3.1 `lxd-imagebuilder`

The reference material in this section provides technical descriptions of `lxd-imagebuilder`.

Actions

```
actions:
    - trigger: <string> # required
      action: |-
        #!/bin/bash
        echo "Run me"
      architectures: <array> # filter
      releases: <array> # filter
      variants: <array> # filter
```

Actions are scripts that are to be run after certain steps during the building process. Each action has two fields, `trigger` and `action`, as well as some filters. The `trigger` field describes the step after which the `action` is to be run. Valid triggers are:

- post-unpack
- post-update
- post-packages
- post-files

The above list also shows the order in which the actions are processed.

After the root file system has been unpacked, all post-unpack actions are run.

After the package manager has updated all packages, (given that packages.update is true), all post-update actions are run. After the package manager has installed the requested packages, all post-packages actions are run. For more on packages, see *packages*.

And last, after the files section has been processed, all post-files actions are run. This action runs only for build-lxc, build-lxd, pack-lxc, and pack-lxd. For more on files, see *generators*.

Command line options

The following are the command line options of lxd-imagebuilder. You can use lxd-imagebuilder to create container and VM images for LXD.

```
$ lxd-imagebuilder
System container and VM image builder for LXD and LXC

Usage:
  lxd-imagebuilder [command]

Available Commands:
  build-dir      Build plain rootfs
  build-lxc      Build LXC image from scratch
  build-lxd      Build LXD image from scratch
  help           Help about any command
  pack-lxc       Create LXC image from existing rootfs
  pack-lxd       Create LXD image from existing rootfs
  repack-windows Repack Windows ISO with drivers included

Flags:
      --cache-dir         Cache directory
      --cleanup           Clean up cache directory (default true)
      --debug             Enable debug output
      --disable-overlay   Disable the use of filesystem overlays
  -h, --help              help for lxd-imagebuilder
  -o, --options           Override options (list of key=value)
  -t, --timeout           Timeout in seconds
      --version           Print version number

Use "lxd-imagebuilder [command] --help" for more information about a command.
```

Filters

Filters can be used to restrict certain sections from being run or being applied. There are three filters, `releases`, `architectures`, and `variants`, and each filter takes a list.

Here's an example:

```
releases:
- v1
- v2
architectures:
- x86_64
variants:
- cloud
```

In the above case, the section will only be applied or run if the release is v1 or v2, the architecture is x86_64 *and* the variant is cloud.

Filters can be applied to each item individually in the lists of following sections:

- files
- sets (packages)
- actions
- repositories

Generators

Generators are used to create, modify or remove files inside the rootfs. Available generators are

- *cloud-init*
- *dump*
- *copy*
- *hostname*
- *hosts*
- *remove*
- *template*
- *lxd-agent*
- *fstab*

In the image definition YAML, they are listed under `files`.

```
files:
    - generator: <string> # which generator to use (required)
      name: <string>
      path: <string>
      content: <string>
      template:
          properties: <map>
          when: <array>
      templated: <boolean>
```

(continues on next page)

(continued from previous page)

```
    mode: <string>
    gid: <string>
    uid: <string>
    pongo: <boolean>
    source: <string>
    architectures: <array> # filter
    releases: <array> # filter
    variants: <array> # filter
```

Filters can be applied to each entry in `files`. Valid filters are `architecture`, `release` and `variant`. See filters for more information.

If `pongo` is `true`, the values of `path`, `content`, and `source` are rendered using Pongo2.

cloud-init

For LXC images, the generator disables cloud-init by disabling any cloud-init services, and creates the file `cloud-init.disable` which is checked by `cloud-init` on startup.

For LXD images, the generator creates templates depending on the provided name. Valid names are `user-data`, `meta-data`, `vendor-data` and `network-config`. The default `path` if not defined otherwise is `/var/lib/cloud/seed/nocloud-net/<name>`. Setting `path`, `content` or `template.properties` will override the default values.

dump

The `dump` generator writes the provided `content` to a file set in `path`. If provided, it will set the `mode` (octal format), `gid` (integer) and/or `uid` (integer).

copy

The `copy` generator copies the file(s) from `source` to the destination `path`. `path` can be left empty and in that case the data will be placed in the same `source` path but inside the container. If provided, the destination `path` will set the `mode` (octal format), `gid` (integer) and/or `uid` (integer). Copying will be done according to the following rules:

- If `source` is a directory, the entire contents of the directory are copied. Only symlinks and regular files are supported.
  - Note 1: The directory itself is not copied, just its contents.
  - Note 2: For files copied, only regular Unix permissions are kept.
- If `source` is a symlink or a regular file, it is copied individually along with its metadata. In this case, if `path` ends with a trailing slash `/`, it will be considered a directory and the contents of `source` will be written at `path`/base(`source`).
- If `path` does not end with a trailing slash, it will be considered a regular file and the contents of `source` will be written at `path`.
- If `path` does not exist, it is created along with all missing directories in its path.
- Multiple `source` resources can be specified using Golang `filepath.Match` regexps. For simplicity they are only allowed in the base name and not in the directory hierarchy. If more than one match is found, `path` will be automatically interpreted as a directory.

hostname

For LXC images, the host name generator writes the LXC specific string `LXC_NAME` to the `hostname` file set in `path`. If the path doesn't exist, the generator does nothing.

For LXD images, the generator creates a template for `path`. If the path doesn't exist, the generator does nothing.

hosts

For LXC images, the generator adds the entry `127.0.0.1 LXC_NAME` to the hosts file set in `path`.

For LXD images, the generator creates a template for the hosts file set in `path`, adding an entry for `127.0.0.1 {{ container.name }}`.

remove

The generator removes the file set in `path` from the container's root file system.

template

This generator creates a custom LXD template. The `name` field is used as the template's file name. The `path` defines the target file in the container's root file system. The `properties` key is a map of the template properties.

The `when` key can be one or more of:

- create (run at the time a new container is created from the image)
- copy (run when a container is created from an existing one)
- start (run every time the container is started)

See Image format in the LXD documentation for more information.

lxd-agent

This generator creates the `systemd` unit files which are needed to start the `lxd-agent` in LXD VMs.

fstab

This generator creates an `/etc/fstab` file which is used for VMs. Its content is:

```
LABEL=rootfs  /         <fs>  <options>  0 0
LABEL=UEFI    /boot/efi vfat  defaults   0 0
```

The file system is taken from the LXD target (see *targets*) which defaults to `ext4`. The options are generated depending on the file system. You cannot override them.

Image

The image section describes the image output.

```
image:
    distribution: <string> # required
    architecture: <string>
    description: <string>
    expiry: <string>
    name: <string>
    release: <string>
    serial: <string>
    variant: <string>
```

The fields `distribution`, `architecture`, `description` and `release` are self-explanatory. If `architecture` is not set, it defaults to the host's architecture.

The `expiry` field describes the image expiry. The format is `\d+(s|m|h|d|w)` (seconds, minutes, hours, days, weeks), and defaults to 30 days (`30d`). It's also possible to define multiple such parts, e.g. `1h 30m 10s`.

The `name` field is used in the LXD metadata as well as the output name for LXD unified tarballs. It defaults to `{{ image.distribution }}-{{ image.release }}-{{ image.architecture_mapped }}-{{ image.variant }}-{{ image.serial }}`.

The `serial` field is the image's serial number. It can be anything and defaults to `YYYYmmdd_HHMM` (date format).

The `variant` field can be anything and is used in the LXD metadata as well as for *filtering*.

Mappings

`mappings` describes an architecture mapping between the architectures from those used in LXD and those used by the distribution. These mappings are useful if you for example want to build a `x86_64` image but the source tarball contains `amd64` as its architecture.

```
mappings:
    architectures: <map>
    architecture_map: <string>
```

It's possible to specify a custom map using the `architectures` field. Here's an example of a custom mapping:

```
mappings:
    architectures:
        i686: i386
        x86_64: amd64
        armv7l: armhf
        aarch64: arm64
        ppc: powerpc
        ppc64: powerpc64
        ppc64le: ppc64el
```

The mapped architecture can be accessed via `Image.ArchitectureMapped` in the code or `image.architecture_mapped` in the definition file.

There are some preset mappings which can be used in the `architecture_map` field. Those are:

- `alpinelinux`

- altlinux
- archlinux
- centos
- debian
- funtoo
- gentoo
- plamolinux
- voidlinux

Package management

Installing and removing packages can be done using the packages section.

```
packages:
    manager: <string> # required
    update: <boolean>
    cleanup: <boolean>
    sets:
        - packages:
            - <string>
            - ...
          action: <string> # required
          architectures: <array> # filter
          releases: <array> # filter
          variants: <array> # filter
          flags: <array> # install/remove flags for just this set
        - ...
    repositories:
        - name: <string>
          url: <string>
          type: <string>
          key: <string>
          architectures: <array> # filter
          releases: <array> # filter
          variants: <array> # filter
        - ...
```

The manager keys specifies the package manager which is to be used. Valid package manager are:

- apk
- apt
- dnf
- egoportage (combination of portage and ego)
- equo
- anise
- opkg
- pacman

- portage
- slackpkg
- xbps
- yum
- zypper

It's also possible to specify a custom package manager. This is useful if the desired package manager is not supported by LXD imagebuilder.

```
packages:
    custom_manager: # required
        clean: # required
            cmd: <string>
            flags: <array>
        install: # required
            cmd: <string>
            flags: <array>
        remove: # required
            cmd: <string>
            flags: <array>
        refresh: # required
            cmd: <string>
            flags: <array>
        update: # required
            cmd: <string>
            flags: <array>
        flags: <array> # global flags for all commands
    ...
```

If `update` is true, the package manager will update all installed packages.

If `cleanup` is true, the package manager will run a cleanup operation which usually cleans up cached files. This depends on the package manager though and is not supported by all.

A set contains a list of `packages`, an `action`, and optional filters. Here, `packages` is a list of packages which are to be installed or removed. The value of `action` must be either `install` or `remove`. If `flags` is specified for a package set, they are appended to the command specific flags, along with any global flags, when calling the `install` or `remove` command. For example, you can define a package set that should be installed with `--no-install-recommends`.

`repositories` contains a list of additional repositories which are to be added. The `type` field is only needed if the package manager supports more than one repository manager. The `key` field is a GPG armored key ring which might be needed for verification.

Depending on the package manager, the `url` field can take the content of a repository file. The following is possible with `yum`:

```
packages:
  manager: yum
  update: false
  repositories:
    - name: myrepo
      url: |-
        [myrepo]
```

(continues on next page)

(continued from previous page)

```
        baseurl=http://user:password@1.1.1.1
        gpgcheck=0
```

Source

In order to create an image, a source must be defined. The source section is defined as follows:

```
source:
    downloader: <string> # required
    url: <string>
    keys: <array>
    keyserver: <string>
    variant: <string>
    suite: <string>
    same_as: <boolean>
    skip_verification: <boolean>
    components: <array>
```

The `downloader` field defines a downloader which pulls a rootfs image which will be used as a starting point. It needs to be one of

- `alpinelinux-http`
- `alt-http`
- `apertis-http`
- `archlinux-http`
- `centos-http`
- `debootstrap`
- `docker-http`
- `fedora-http`
- `funtoo-http`
- `gentoo-http`
- `nixos-http`
- `openeuler-http`
- `opensuse-http`
- `openwrt-http`
- `oraclelinux-http`
- `sabayon-http`
- `rootfs-http`
- `ubuntu-http`
- `voidlinux-http`

The `url` field defines the URL or mirror of the rootfs image. Although this field is not required, most downloaders will need it. The `rootfs-http` downloader also supports local image files when prefixed with `file://`, e.g. `url: file:///home/user/image.tar.gz` or `url: file:///home/user/image.squashfs`.

The `keys` field is a list of GPG keys. These keys can be listed as fingerprints or armored keys. The latter has the advantage of not having to rely on a key server to download the key from. The keys are used to verify the downloaded rootfs tarball if downloaded from a insecure source (HTTP).

The `keyserver` defaults to `hkps.pool.sks-keyservers.net` if none is provided.

The `variant` field is only used in a few distributions and defaults to `default`. Here's a list downloaders and their possible variants:

- `centos-http`: `minimal`, `netinstall`, `LiveDVD`
- `debootstrap`: `default`, `minbase`, `buildd`, `fakechroot`
- `ubuntu-http`: `default`, `core`
- `voidlinux-http`: `default`, `musl`

All other downloaders ignore this field.

The `suite` field is only used by the `debootstrap` downloader. If set, `debootstrap` will use `suite` instead of `image.release` as its first positional argument.

If the `same_as` field is set, LXD imagebuilder creates a temporary symlink in `/usr/share/debootstrap/scripts` which points to the `same_as` file inside that directory. This can be used if you want to run `debootstrap foo` but `foo` is missing due to `debootstrap` not being up-to-date.

If `skip_verification` is true, the source tarball is not verified.

If the `components` field is set, `debootstrap` will use packages from the listed components.

If a package set has the `early` flag enabled, that list of packages will be installed while the source is being downloaded. (Note that `early` packages are only supported by the `debootstrap` downloader.)

Targets

The target section is for target dependent files.

```
targets:
    lxc:
        create_message: <string>
        config:
            - type: <string>
              before: <uint>
              after: <uint>
              content: <string>
            - ...
    lxd:
        vm:
            size: <uint>
            filesystem: <string>
```

LXC

The `create_message` field is a string which is displayed after new LXC container has been created. This string is rendered using Pongo2 and can include various fields from the definition file, e.g. `{{ image.description }}`.

`config` is a list of container configuration options. The `type` must be `all`, `system` or `user`.

The keys `before` and `after` are used for compatibility. Currently, the maximum value for compatibility is 5. If your desired compatibility level is 3 for example, you would use `before: 4` and `after: 2`.

`content` describes the configuration which is to be written to the configuration file.

LXD

Valid keys are `size` and `filesystem`. The former specifies the VM image size in bytes. The latter specifies the root partition file system. It currently supports `ext4` and `btrfs`.

2.3.2 simplestream-maintainer

The reference material in this section provides technical descriptions of `lxd-imagebuilder`.

Command line options

The following are the command line options of `simplestream-maintainer`.

```
$ simplestream-maintainer
Simplestream server maintainer

Usage:
  simplestream-maintainer [command]

Commands:
  build       Build simplestream index on the given path
  prune       Prune product versions

Other Commands:
  completion  Generate the autocompletion script for the specified shell
  help        Help about any command

Flags:
  -h, --help               help for simplestream-maintainer
      --logformat string   Log format (default "text")
      --loglevel string    Log level (default "info")
      --timeout uint       Timeout in seconds
  -v, --version            version for simplestream-maintainer
```

Simple streams directory structure

`simplestream-maintainer` is a CLI tool for building a simple streams index and product catalog, and removing expired or invalid product versions.

Terminology

- `Stream`: Represents a directory that contains image builds.
- `Product`: Represents a unique image within a single stream. Its ID is determined by the directory structure as `<distro>/<release>/<arch>/<variant>`.
- `ProductVersion`: Represents a version (build) of a specific image. A single product can contain one or more versions. While the version name can be custom, it should allow sorting by time. A good example is a timestamp of the image build.
- `ProductCatalog`: Represents all products from a specific stream.
- `Index`: Contains a list of product catalogs and their products.

Directory structure

For easier representation, here is an example of the simple streams directory structure.

```
.
├── images                 // Stream name.
│   └── ubuntu             // Distribution.
│       └── noble          // Release.
│           └── amd64      // Architecture.
│               └── cloud  // Variant.
│                   ├── 20240505_1212       // Version name.
│                   │   ├── lxd.tar.xz      // Metadata file.
│                   │   ├── disk.qcow2      // Virtual machine rootfs.
│                   │   ├── rootfs.squashfs // Container rootfs.
│                   │   └── images.yaml     // Optional image config.
│                   ├── 20240506_1212
│                   │   ├── lxd.tar.xz
│                   │   ├── disk.qcow2
│                   │   ├── rootfs.squashfs
│                   │   ├── disk.20240505_1212.qcow2.vcdiff // Virtual machine delta␣
↪file from previous version (20240505_1212).
│                   │   ├── rootfs.20240505_1212.vcdiff     // Container delta file from␣
↪previous version (20240505_1212).
│                   │   └── images.yaml
│                   └── 20240507_1212
│                       └── ...
├── images-daily
│   └── ...
├── images-minimal
│   └── ...
└── streams
    ├── v1
    │   ├── index.json
```

(continues on next page)

(continued from previous page)

```
    │   ├── images.json
    │   ├── images-daily.json
    │   └── images-minimal.json
    └── v2
        └── ...
```

Root directory

In the above directory structure, the simple streams *root directory* is represented with a `.` (dot). The CLI tool expects this path to be provided as an argument for each command, as it will operate exclusively within that directory.

Stream directory

The *stream directory* represents the directory within the simple streams root directory. It is expected to contain built images with the following directory structure:

```
<rootDir>
└── <stream>
    └── <distro>
        └── <release>
            └── <arch>
                └── <variant>
                    └── <version>
```

For example:

```
Path:    images/ubuntu/noble/amd64/cloud/v1
---
stream:  images
distro:  ubuntu
release: noble
arch:    amd64
variant: cloud
version: v1
```

Product version

```
...
<version>
├── lxd.tar.xz
├── disk.qcow2
├── rootfs.squashfs
└── images.yaml
```

Each `<version>` directory is considered *complete* if it contains `lxd.tar.xz` (image metadata) and at least one rootfs file (`*.squashfs` and/or `*.qcow2`).

In addition, hidden versions (prefixed with the dot `.<version>`) are treated as incomplete. This allows you to first push the images to the server into a hidden directory, and only once they are fully uploaded unhide the directory. This approach prevents partially uploaded files from being included in the product catalog.

Simple streams configuration

A product version can contain an optional `images.yaml` configuration file. This file can contain additional image information, such as release aliases or image requirements which cannot be parsed purely from the directory structure.

All simple streams related configuration is located within a `simplestream` field, which currently supports:

- `distro_name` - Name of the distribution that is shown when listing images in LXD. It defaults to the distribution name parsed from the directory structure.
- `release_aliases` - A map of the distribution release and a comma-delimited string of release aliases.
- `requirements` - A list of image requirements with optional filters.

> **Note**
>
> The configuration file is always parsed from the last product version (alphabetically sorted).

Example for the distribution name:

```
simplestream:
  distro_name: Ubuntu Core
```

Example for release aliases:

```
simplestream:
  release_aliases:
    jammy: 22.04     # Single alias.
    noble: 24.04,24  # Multiple aliases.
```

Example for requirements:

```
simplestream:
  requirements:

  # Applied to all images (no filters).
  - requirements:
      secure_boot: false

  # Applied to images that match the filters.
  - requirements:
      nesting: true
    releases:
    - noble
    architectures:
    - amd64
    variant:
    - default
    - desktop
```