Flatpak Documentation

Release

Flatpak Team

Jul 18, 2017

Contents

These docs cover everything you need to know to build and distribute applications as Flatpaks. They contain a high-level introduction to Flatpak, tutorials and essential information on how to develop, build and distribute applications.

The docs are primarily intended for application developers and distributors. Their content is also relevant to those who have a general interest in Flatpak.

If you are looking for information about how to install and use Flatpak applications, please refer to the Flatpak website.

CHAPTER 1

Contents

Introduction to Flatpak

Flatpak is a technology for building, distributing, installing and running applications. It is primarily targeted at the Linux desktop, although it can also be used as the basis for application distribution in other contexts, such as embedded systems.

Flatpak has been designed and implemented with a number of goals:

- Allow applications to be installed on any Linux distribution.
- Provide consistent environments for applications, to facilitate testing and reduce bugs.
- Decouple applications from the operating system, so that applications don't depend on specific versions of each distribution.
- Allow applications to bundle their own dependencies, so that they can use libraries that aren't provided by a Linux distribution, and so they can depend on specific versions or even patched versions of a library.
- Increase the security of Linux desktops, by isolating applications in sandboxes.

Flatpak makes it easy to take advantage of these features. If you haven't already, it is recommended that you try the hello world example, as a way of getting started.

More information about Flatpak can be found on flatpak.org.

How it works

Flatpak can be understood through a small number of key concepts. These also help to explain how it differs from traditional software packages.

Runtimes

Runtimes provide the basic dependencies that are used by applications. Various runtimes are available, from more minimal (but more stable) Freedesktop runtimes, to larger runtimes produced by desktops like GNOME or KDE. (The runtimes page on flatpak.org provides an overview of the runtimes that are currently available.)

Each application must be built against a runtime, and this runtime must be installed on a host system in order for the application to run. Users can install multiple different runtimes at the same time, including different versions of the same runtime.

Tip: Each runtime can be thought of as a `/usr` filesystem. Indeed, when an application is run, its runtime is mounted at `/usr`.

Bundled libraries

If an application requires any dependencies that aren't in its runtime, they can be bundled along with the application itself. This allows applications to use dependencies that aren't available in a distribution, or to use a different version of a dependency from the one that's installed on the host.

SDKs (Software Developer Kits)

An SDK is a runtime that includes the 'devel' parts which are not needed at runtime, such as build and packaging tools, header files, compilers and debuggers. Each application is built against an SDK, which is paired with a runtime (this is the runtime that will be used by the application at runtime).

Extensions

An extension is an optional add-on for a runtime or application. They are most commonly used to split out translations and debug info from runtimes. For example, `org.freedesktop.Platform.Locale` can be added to the `org.freedesktop.Platform` runtime in order to add translations.

Sandboxes

With Flatpak, each app is built and run in an isolated environment. By default, the application can only 'see' itself and its runtime. Access to user files, network, graphics sockets, subsystems on the bus and devices have to be explicitly granted. (As will be described later, Flatpak provides several ways to do this.) Access to other things, such as other processes, is deliberately not possible.

The flatpak command

`flatpak` is the command that is used to install, remove and update runtimes and applications. It can also be used to view what is currently installed, and has commands for building and distributing application bundles. `flatpak --help` provides a full list of available commands.

Most flatpak commands are performed system-wide by default. To perform a command for the current user only, use the `--user` option. This allows runtimes and application bundles to be installed per-user, for instance.

For more information on flatpak commands, see the *Command Reference*

Identifiers

Flatpak identifies each application, runtime and SDK using a unique name, which is sometimes used as part of a name/arch/branch triple.

Naming

Flatpak names take the form of an inverse DNS address, such as `com.company.App`. The final segment of this address is the object's name, and the preceding part is the domain that it belongs to. In order to prevent name conflicts, this domain should correspond to a DNS registered address. This means using a domain from a website, either for an application or an organization. For instance, if application `App` has its own website at `app.com`, its Flatpak name would be `com.app.App`. Multiple applications can belong to the same domain, such as `org.organization.App1` and `org.organization.App2`.

If you do not have a registered domain for your application, it is easy to use a third party website to get one. For example, Github allows the creation of personal pages that can be used for this purpose. Here, a personal namespace of `name.github.io` could be used as the basis of application identifier `io.github.name.App`.

If an application provides a D-Bus service, the D-Bus service name is expected to be the same as the application name.

Identifier triples

Many flatpak commands only require the name of an application, runtime or SDK. However, in some circumstances it is also necessary to specify the architecture and branch (branches allow a particular version to be specified). This is done using a name/arch/branch triple. For example: `org.gnome.Sdk/x86_64/3.14` or `org.gnome.Builder/i386/master`.

Under the hood

Flatpak uses a number of pre-existing technologies. It generally isn't necessary to be familiar with these in order to use Flatpak, although in some cases it might be useful. They include:

- The bubblewrap utility from Project Atomic, which lets unprivileged users set up and run containers, using kernel features such as:
  - Cgroups
  - Namespaces
  - Bind mounts
  - Seccomp rules
- systemd to set up cgroups for sandboxes
- D-Bus, a well-established way to provide high-level APIs to applications
- The OCI format from the Open Container Initiative, as a convenient transport format for single-file bundles
- The OSTree system for versioning and distributing filesystem trees
- Appstream metadata, to allow Flatpak applications to show up nicely in software-center applications

Elements of a Flatpak Application

Flatpak expects applications to follow standard Linux desktop conventions. These are supplemented with a small number of Flatpak-specific elements that are used to distribute, install and run applications.

Standard application elements

The following are some of the Linux desktop conventions that are supported and expected by Flatpak. Application developers are encouraged to use them.

- AppData, for providing application information, such as descriptions and screenshots, that is used by app stores
- Application icons, as specified by the Freedesktop icon theme specification
- D-Bus, for interaction with the host
- Desktop files, for providing basic information about the application
- PulseAudio, for sound
- X11 and Wayland, for display

Application structure

When an application is built using flatpak, it is outputted with the following structure:

- `metadata` - a keyfile which provides information about the application
- `/files` - the files that make up the application, include source code and application data
- `/files/bin` - application binaries
- `/export` - files which the host environment needs access to, such as the application's AppData, .desktop file, icon and D-Bus service files

All the files in the export directory must have the application ID as their prefix. For example:

- `org.gnome.App.appdata.xml`
- `org.gnome.App.desktop`
- `org.gnome.App.png`
- `org.gnome.App.service`

Naming files in this way prevents naming conflicts and ensures that system installed applications aren't overwritten.

To name exported files in this way, either rename the relevant source files or use flatpak-builder to rename the files at build time (this is explained in more detail in the section on flatpak-builder).

Metadata files

An application's `metadata` file provides information that allows flatpak to set up the sandbox for running the application. A typical metadata file looks like this:

```
[Application]
name=org.gnome.gedit
runtime=org.gnome.Platform/x86_64/3.22
sdk=org.gnome.Sdk/x86_64/3.22
command=gedit
```

```
[Context]
shared=ipc;network;
sockets=x11;wayland;pulseaudio;
devices=dri;
filesystems=host;

[Environment]
GEDIT_FOO=bar

[Session Bus Policy]
org.extra.name=talk
org.other.name=own
```

This specifies the name of the application, the runtime it requires, the SDK that it was built against and the command used to run it. It also specifies file and device access, sets certain environment variables (inside the application's sandbox, of course), and how it connects to the session bus. Details on how to change these metadata parameters are included in subsequent sections.

Note: While it is most common to encounter metadata files for applications, runtimes and extensions also have them.

Getting Setup

Getting setup to build Flatpaks is quick and easy. First, it is necessary to have the `flatpak` and `flatpak-builder` packages installed on your system. These are available for most distributions, and the Flatpak website provides details on how to get them.

Once flatpak has been installed, it is necessary to pick a runtime and install it, along with the matching SDK.

Installing an SDK

An SDK is a special type of runtime that is used to build applications. Typically, an SDK is paired with a runtime that will be used by the application at runtime. For example the GNOME 3.22 SDK is used to build applications that use the GNOME 3.22 runtime.

The Flatpak website provides a list of the available runtimes. Once you have decided which one to use, getting setup is just a matter of installing it and its SDK.

The examples in the rest of the Flatpak documentation use the GNOME 3.22 runtime and SDK. If you haven't installed these already, download the repository GPG key and then add the repository that contains the runtime and SDK:

```
$ flatpak remote-add --from gnome https://sdk.gnome.org/gnome.flatpakrepo
```

You can now download and install the runtime and SDK:

```
$ flatpak install gnome org.gnome.Platform//3.22 org.gnome.Sdk//3.22
```

This same procedure can be used to install any other runtime and SDK.

Taking a look around

If this is the first time you've used Flatpak, it is a good time to try installing an application and having a look 'under the hood'. To do this, you need to add a repository that contains applications. We can do this using the gnome-apps repository to install gedit:

```
$ flatpak remote-add --from gnome-apps https://sdk.gnome.org/gnome-apps.flatpakrepo
$ flatpak install gnome-apps org.gnome.gedit
```

You can now use the following command to get a shell in the 'devel sandbox':

```
$ flatpak run --devel --command=bash org.gnome.gedit
```

This gives you an environment which has the application bundle mounted in `/app`, and the SDK it was built against mounted in `/usr`. You can explore these two directories to see what a typical flatpak looks like, as well as what is included in the SDK.

Building Simple Apps

The `flatpak` utility provides a simple set of commands for building and distributing applications. These allow creating new Flatpaks, into which new or existing applications can be built.

This section describes how to build a simple application which doesn't require any additional dependencies outside of the runtime it is built against. In order to complete the examples, you should have completed the steps in Getting Setup first.

Creating an app

To create an application, the first step is to use the `build-init` command. This creates a directory into which an application can be built, which contains the correct directory structure and a metadata file which contains information about the app. The format for build-init is:

```
$ flatpak build-init DIRECTORY APPNAME SDK RUNTIME [BRANCH]
```

- DIRECTORY is the name of the directory that will be created to contain the application
- APPNAME is the D-Bus name of the application
- SDK is the name of the SDK that will be used to build the application
- RUNTIME is the name of the runtime that will be required by the application
- BRANCH is typically the version of the SDK and runtime that will be used

For example, to build the GNOME Dictionary application using the GNOME 3.22 SDK, the command would look like:

```
$ flatpak build-init dictionary org.gnome.Dictionary org.gnome.Sdk org.gnome.Platform␣
↪3.22
```

You can try this command now. In the next step we will build an application inside the resulting dictionary directory.

Building

`flatpak build` is used to build an application using an SDK. This command is used to provide access to a sandbox. For example, the following will create a file inside the appdir sandbox (in the files directory):

```
$ flatpak build dictionary touch /app/some_file
```

(It is best to remove this file before continuing.)

The build command allows existing applications that have been made using the traditional configure, make, make install routine to be built inside a flatpak. You can try this using GNOME Dictionary. First, download the source files, extract them and switch to the resulting directory:

```
$ wget https://download.gnome.org/sources/gnome-dictionary/3.20/gnome-dictionary-3.20.
↪0.tar.xz
$ tar xvf gnome-dictionary-3.20.0.tar.xz
$ cd gnome-dictionary-3.20.0/
```

Then you can use the build command to build and install the source inside the dictionary directory that was previously made:

```
$ flatpak build ../dictionary ./configure --prefix=/app
$ flatpak build ../dictionary make
$ flatpak build ../dictionary make install
$ cd ..
```

Since these are run in a sandbox, the compiler and other tools from the SDK are used to build and install, rather than those on the host system.

Completing the build

Once an application has been built, the `build-finish` command needs to be used to specify access to different parts of the host, such as networking and graphics sockets. This command is also used to specify the command that is used to run the app (done by modifying the metadata file), and to create the application's exports directory. For example:

```
$ flatpak build-finish dictionary --socket=x11 --share=network --command=gnome-
↪dictionary
```

At this point you have successfully built a flatpak and prepared it to be run. To test the app, you need to export the Dictionary to a repository, add that repository and then install and run the app:

```
$ flatpak build-export repo dictionary
$ flatpak --user remote-add --no-gpg-verify --if-not-exists tutorial-repo repo
$ flatpak --user install tutorial-repo org.gnome.Dictionary
$ flatpak run org.gnome.Dictionary
```

This exports the app, creates a repository called tutorial-repo, installs the Dictionary application in the per-user installation area and runs it.

Flatpak Builder

Most applications require additional dependencies that aren't provided by their runtimes. Flatpak allows these dependencies to be bundled as part of the application itself. In order to do this, each dependency must be built inside the

application build directory. The `flatpak-builder` tool automates this multi-step build process, making it possible to build all application modules with a single command.

flatpak-builder expects modules to be built in the standard manner by following what is called the Build API. This requires modifying modules to follow the build API, if they don't already.

All json entities are explained in the man page of `flatpak-builder`.

Manifests

The input to flatpak-builder is a JSON file that describes the parameters for building an application, as well as each of the modules to be bundled. This file is called the manifest. Module sources can be of several types, including `.tar` or `.zip` archives, Git or Bzr repositories, patch files or shell commands that are run.

The GNOME Dictionary manifest is short, because the only module is the application itself:

```
{
  "app-id": "org.gnome.Dictionary",
  "runtime": "org.gnome.Platform",
  "runtime-version": "3.22",
  "sdk": "org.gnome.Sdk",
  "command": "gnome-dictionary",
  "finish-args": [
     "--socket=x11",
     "--share=network"
  ],
  "modules": [
    {
      "name": "gnome-dictionary",
      "sources": [
        {
          "type": "archive",
          "url": "https://download.gnome.org/sources/gnome-dictionary/3.20/gnome-
↪dictionary-3.20.0.tar.xz",
          "sha256": "efb36377d46eff9291d3b8fec37baab2355f9dc8bc7edb791b6a625574716121"
        }
      ]
    }
  ]
}
```

As can be seen, this manifest includes basic information about the application before specifying a single .tar file to be downloaded and built. More complex manifests include a sequence of modules.

Cleanup

After building has taken place, flatpak-builder performs a cleanup phase. This can be used to remove headers and development documentation, among other things. Two properties in the manifest file are used for this. First, a list of filename patterns can be included:

```
"cleanup": [ "/include", "/bin/foo-*", "*.a" ]
```

The second cleanup property is a list of commands that are run during the cleanup phase:

```
"cleanup-commands": [ "sed s/foo/bar/ /bin/app.sh" ]
```

Cleanup properties can be set on a per-module basis, in which case only filenames that were created by that particular module will be matched.

File renaming

Files that are exported by a flatpak must be prefixed using the application ID. If an application's source files are not named using this convention, flatpak-builder allows them to be renamed as part of the build process. To rename application icons, desktop files and AppData files, use the `rename-icon`, `rename-desktop-file` and `rename-appdata` properties.

Splitting things up

By default, flatpak-builder splits off translations and debug information into separate .Locale and .Debug extensions. These 'standard' extension points are then added to the application's metadata file. You can turn this off with the `separate-locales` and `no-debuginfo` keys, but there shouldn't be any reason for it.

When flatpak-builder exports the build into a repository, it automatically includes the .Locale and .Debug extensions. If you do the exporting manually, don't forget to include them.

Example

To try flatpak-builder yourself, create a file called `org.gnome.Dictionary.json` and paste the Dictionary manifest JSON from above into it. Then run the following command:

```
$ flatpak-builder --repo=repo dictionary2 org.gnome.Dictionary.json
```

This will:

- Create a new directory called dictionary2 (equivalent to using *flatpak build-init*)
- Download and verify the Dictionary source code
- Build and install the source code, using the SDK rather than the host system
- Finish the build, by setting permissions (in this case giving access to X and the network)
- Create a new repository called repo (if it doesn't exist) and export the resulting build into it

flatpak-builder will also do some other useful things, like creating a separately installable debug runtime (called *org.gnome.Dictionary.Debug* in this case) and a separately installable translation runtime (called `org.gnome.Dictionary.Locale`).

If you completed the tutorial in Building Simple Apps, you can update the Dictionary application you installed with the new version that was built and exported by flatpak-builder:

```
$ flatpak --user update org.gnome.Dictionary
```

To check that the application has been successfully updated, you can compare the sha256 commit of the installed app with the commit ID that was printed by flatpak-builder:

```
$ flatpak info org.gnome.Dictionary
$ flatpak info org.gnome.Dictionary.Locale
```

And finally, you can run the new version of the Dictionary app:

```
$ flatpak run org.gnome.Dictionary
```

Example manifests

A complete manifest for GNOME Dictionary built from Git is available, in addition to manifests for a range of other GNOME applications.

Working with the Sandbox

One of Flatpak's main goals is to increase the security of desktop systems by isolating applications from one another. This is done using sandboxing and it means that, by default, a Flatpak has extremely limited access to the host environment. This includes:

- No access to any host files except the runtime, the app and `~/.var/app/$APPID`. Only the last of these is writable.
- No access to the network.
- No access to any device nodes (apart from `/dev/null`, etc).
- No access to processes outside the sandbox.
- Limited syscalls. For instance, apps can't use nonstandard network socket types or ptrace other processes.
- Limited access to the session D-Bus instance - an app can only own its own name on the bus.
- No access to host services like X, system D-Bus, or PulseAudio.

Most applications will need access to some of these resources in order to be useful, and Flatpak provides a number of ways to give an application access to them.

While there are no restrictions on which sandbox permissions an application can use, as good practice, it is recommended to use the minimum number of as permissions possible. Certain permissions, such as blanket access to the system bus (using the `--socket=system-bus` option) are strongly discouraged.

Configuring sandbox permissions

Using the `build-finish` command is the simplest way to configure sandbox permissions. As was seen in a previous example, this can be used to add access to graphics sockets and the network:

```
$ flatpak build-finish dictionary2 --socket=x11 --share=network --command=gnome-
↪dictionary
```

These arguments translate into several properties in the application's metadata file:

```
[Application]
name=org.gnome.Dictionary
runtime=org.gnome.Platform/x86_64/3.22
sdk=org.gnome.Sdk/x86_64/3.22
command=gnome-dictionary

[Context]
shared=network;
sockets=x11;
```

build-finish allows a whole range of resources to be added to an application. These options can also be passed to flatpak-builder as `finish-args` properties.

The table at the bottom of this page provides an overview of many sandbox permissions. The full list can also be viewed using `flatpak build-finish --help`.

Note: Until a sandbox-compatible backend is available, access to dconf needs to be enabled using the following options:

```
--filesystem=xdg-run/dconf
--filesystem=~/.config/dconf:ro
--talk-name=ca.desrt.dconf
--env=DCONF_USER_CONFIG_DIR=.config/dconf
```

Portals

Portals are a mechanism through which applications can interact with the host environment from within a sandbox. In this way, they give additional abilities to interact with data, files and services without the need to add sandbox permissions.

Interface toolkits can implement transparent support for portals. If an application uses one of these toolkits, there is no additional work required to access them.

Examples of capabilities that can be accessed through portals include:

- Inhibit the user session from ending, suspending, idling or getting switched away
- Network status information
- Notifications
- Open a URI
- Open files with a native file chooser dialog
- Printing
- Screenshots

Applications that aren't using a toolkit with support for portals can refer to the xdg-desktop-portal API documentation for information on how to access them.

Overriding sandbox permissions

When developing an application, it can sometimes be useful to override a Flatpak's sandbox configuration. There are several ways to do this. One is to override them using `flatpak run`, which accepts the same parameters as `build-finish`. For example, this will let the Dictionary application see your home directory:

```
$ flatpak run --filesystem=home --command=ls org.gnome.Dictionary ~/
```

`flatpak override` can also be used to permanently override an application's permissions:

```
$ flatpak --user override --filesystem=home org.gnome.Dictionary
$ flatpak run --command=ls org.gnome.Dictionary ~/
```

It is also possible to remove permissions using the same method. You can use the following command to see what happens when access to the filesystem is removed, for example:

```
$ flatpak run --nofilesystem=home --command=ls org.gnome.Dictionary ~/
```

Useful sandbox permissions

Flatpak provides an array of options for controlling sandbox permissions. The following are some of the most useful:

–filesystem=host	Access all files
–filesystem=home	Access the home directory
–filesystem=home:ro	Access the home directory, read-only
–filesystem=/some/dir –filesystem=~/other/dir	Access paths
–filesystem=xdg-download	Access the XDG download directory
–nofilesystem=...	Undo some of the above
–socket=x11 –share=ipc	Show windows using X11[1]
–device=dri	OpenGL rendering
–socket=wayland	Show windows using Wayland
–socket=pulseaudio	Play sounds using PulseAudio
–share=network	Access the network[2]
–talk-name=org.freedesktop.secrets	Talk to a named service on the session bus
–system-talk-name=org.freedesktop.GeoClue2	Talk to a named service on the system bus
–socket=system-bus	Unlimited access to all of D-Bus

Distributing Applications

Flatpak provides several ways to distribute applications. The primary method is to host a repository. This is relatively simple (although there are some important details to be aware of) and allows application updates to be distributed.

It is also possible to distribute Flatpaks as single file bundles, which can be useful in some situations.

Hosting a repository

The previous sections of this guide describe how to generate repositories using `build-export` or `flatpak-builder`. The resulting OSTree repository can be hosted on a web server for consumption by users.

Important details

OSTree repositories use archive-z2, meaning that they contain a single file for each file in the application. This means that pull operations will do a lot of HTTP requests. Since new requests are slow, it is important to enable HTTP keep-alive on the web server.

OSTree supports something called static deltas. These are single files in the repo that contains all the data needed to go between two revisions (or from nothing to a revision). Creating such deltas will take up more space on the server, but will make downloads much faster. This can be done with the `build-update-repo --generate-static-deltas` option.

[1] `-share=ipc` means that the sandbox shares IPC namespace with the host. This is not necessarily required, but without it the X shared memory extension will not work, which is very bad for X performance.

[2] Giving network access also grants access to all host services listening on abstract Unix sockets (due to how network namespaces work), and these have no permission checks. This unfortunately affects e.g. the X server and the session bus which listens to abstract sockets by default. A secure distribution should disable these and just use regular sockets.

GPG signatures

OSTree uses GPG to verify the identity of repositories. This requires that every commit to a repository uses a GPG signature, as well as when repository summary files are modified.

To do this, a GPG key needs to be passed to the `build-update-repo` and `build-export` commands, as well as `flatpak-builder` if it is being used to modify or create a repository. (If you don't already have a key, it is easy to generate one.) For example:

```
$ flatpak build-export --gpg-sign=KEYID --gpg-homedir=PATH REPOSITORY DIRECTORY
```

Here `--gpg-homedir` is optional, and allows specifying the home directory of the key to be used.

Though it generally isn't recommended, it is possible to disable GPG verification of OSTree repositories. To do this, the `--no-gpg-verify` option can be used when a remote is added. GPG verification can also be disabled on an existing remote using `flatpak remote-modify`.

Note that it is necessary to become root in order to update a remote that does not have GPG verification enabled.

Referring to repositories

A convenient way to point users to the repository containing your application is to provide a `.flatpakrepo` file that they can download and install. To install a `.flatpakrepo` file manually, use the command:

```
$ flatpak remote-add --from foo.flatpakrepo
```

A typical `.flatpakrepo` file looks like this:

```
[Flatpak Repo]
Title=GEdit
Url=http://sdk.gnome.org/repo-apps/
GPGKey=mQENBFUUCGcBCAC/K9WeV4xCaKr3...
```

If your repository contains just a single application, it may be more convenient to use a .flatpakref file instead, which contains enough information to add the repository and install the application at the same time. To install a `.flatpakref` file manually, use the command:

```
$ flatpak install --from foo.flatpakref
```

A typical `.flatpakref` file looks like this:

```
[Flatpak Ref]
Title=GEdit
Name=org.gnome.gedit
Branch=stable
Url=http://sdk.gnome.org/repo-apps/
IsRuntime=False
GPGKey=mQENBFUUCGcBCAC/K9WeV4xCaKr3...
RuntimeRepo=https://sdk.gnome.org/gnome.flatpakrepo
```

Note that the GPGKey key in these files contains the base64-encoded GPG key, which you can get with the following command:

```
$ base64 --wrap=0 < foo.gpg
```

Single-file bundles

Hosting a repository is the preferred way to distribute an application, but sometimes a single-file bundle that you can make available from a website or send as an email attachment is more convenient. Flatpak supports this with the `build-bundle` and `build-import-bundle` commands to convert an application in a repository to a bundle and back:

```
$ flatpak build-bundle [OPTION...] LOCATION FILENAME NAME [BRANCH]
$ flatpak build-import-bundle [OPTION...] LOCATION FILENAME
```

For example, to create a bundle named *dictionary.flatpak* containing the GNOME dictionary app from the repository at ~/repositories/apps, run:

```
$ flatpak build-bundle ~/repositories/apps dictionary.flatpak org.gnome.Dictionary
```

To import the bundle into a repository on another machine, run:

```
$ flatpak build-import-bundle ~/my-apps dictionary.flatpak
```

Note that bundles have some drawbacks, compared to repositories. For example, distributing updates is much more convenient with a hosted repository, since users can just run `flatpak update`.

Command Reference