Flatpak Documentation

Release

Flatpak Team

Mar 17, 2018

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

1.1 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.

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

1.1.1 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. (See *Available Runtimes* for 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.

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.

By necessity, some resources that are inside the sandbox need to be exposed outside, to be used by the host system. These are known as "exports", since they are files that are exported out of the sandbox, and include things like the application's `.desktop` file and icon.

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.

1.1.2 The flatpak command

`flatpak` is the command that is used to find, install and remove applications. `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 *Flatpak Command Reference*

1.1.3 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`.

1.1.4 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

1.2 Getting Started

First steps with Flatpak, including installation, basic usage, and building your first app.

1.2.1 Setup

Install flatpak and flatpak-builder

First, it is necessary to have the `flatpak` and `flatpak-builder` packages installed on your system. The Flatpak website provides details on how to install flatpak. `flatpak-builder` is typically found as a package from the same source as `flatpak` itself.

Add the flathub repository

Flathub is the main Flatpak repository and contains the runtimes and SDKs that will be needed to run and build apps. If you haven't added it already, it can be setup with the following command:

```
$ flatpak remote-add --if-not-exists flathub https://dl.flathub.org/repo/flathub.
↪flatpakrepo
```

1.2.2 Using Flatpak

This page provides an introduction to the most common commands needed to use Flatpak. It is not intended to be exhaustive or to cover all the options for each command (a full list of all the commands and their options can be found in the *Flatpak Command Reference*.

Note: Flatpak commands can be run either per-user or system-wide. All the examples on this page use the default system-wide behavior, which is generally the right option to use.

Remotes

Remotes are the repositories from which applications and runtimes can be installed.

List remotes

To list the remotes that you have configured on your system, run:

```
$ flatpak remotes
```

This gives a list of the existing remotes that have been added. The list indicates whether each remote has been added per-user or system-wide.

Add a remote

Adding a remote allows you to search and list its contents, and to install applications from it. The most convenient way to add a remote is by using a `.flatpakrepo` file, which includes both the details of the remote and its GPG key:

```
$ flatpak remote-add --if-not-exists flathub https://dl.flathub.org/repo/flathub.
↪flatpakrepo
```

Here, `flathub` is the local name that is given to the remote. The URL points to the remote's `.flatpakrepo` file. `--if-not-exists` stops the command from producing an error if the remote already exists.

Remove a remote

To remove a remote, run:

```
$ flatpak remote-delete flathub
```

In this case, `flathub` is the remote's local name.

Installing applications

Search

Applications can be found in any of your remotes using the `search` command. For example:

```
$ flatpak search gimp
```

Search will return any applications matching the search terms. Each search result includes the application ID and the remote that the application is in. In this example, the search term is `gimp`.

Note: Prior to Flatpak 0.11.1, it was necessary to manually update the metadata for your remotes before search will work. This can be done by either running `flatpak update` or `flatpak update --appstream`.

Install applications

To install an application, run:

```
$ flatpak install flathub org.gimp.GIMP
```

Here, `flathub` is the name of the remote the application is to be installed from, and `org.gimp.GIMP` is the ID of the application.

Sometimes, an application will require a particular runtime, and this will be installed prior to the application.

The details of the application to be installed can also be provided by a `.flatpakref` file, which can be either remote or local. To specify a `.flatpakref` instead of manually providing the remote and application ID, run:

```
$ flatpak install https://flathub.org/repo/appstream/org.gimp.GIMP.flatpakref
```

If the `.flatpakref` file specifies that the application is to be installed from a remote that hasn't already been added, you will be asked whether to add it before the application is installed.

Running applications

Once an application has been installed, it can be launched using the `run` command and its application ID:

```
$ flatpak run org.gimp.GIMP
```

Managing your applications

Updating

To update all your installed applications and runtimes to the latest version, run:

```
$ flatpak update
```

List installed applications

To list the applications and runtimes you have installed, run:

```
$ flatpak list
```

Alternatively, to just list installed applications, run:

```
$ flatpak list --app
```

Remove an application

To remove an application, run:

```
$ flatpak uninstall org.gimp.GIMP
```

1.2.3 Building your first Flatpak

This tutorial provides a quick introduction to building Flatpaks. In it, you will learn how to create a basic Flatpak application, which can be installed and run.

The *Setup* page should be followed before doing this tutorial.

1. Install a runtime and the matching SDK

Flatpak requires every app to specify a runtime that it uses for its basic dependencies. Each runtime has a matching SDK (Software Development Kit), which contains all the things that are in the runtime, plus headers and development tools. This SDK is required to build apps for the runtime.

In this tutorial we will use the Freedesktop 1.6 runtime and SDK. To install these, run:

```
$ flatpak install flathub org.freedesktop.Platform//1.6 org.freedesktop.Sdk//1.6
```

2. Create the app

The app that is going to be created for this tutorial is a simple script. To create it, copy the following:

```
#!/bin/sh
echo "Hello world, from a sandbox"
```

Now paste this into an empty file and save it as *hello.sh*.

3. Add a manifest

Each Flatpak is built using a manifest file which provides basic information about the application and instructions for how it is to be built. To add a manifest to the hello world app, add the following to an empty file:

```
{
    "app-id": "org.flatpak.Hello",
    "runtime": "org.freedesktop.Platform",
    "runtime-version": "1.6",
    "sdk": "org.freedesktop.Sdk",
    "command": "hello.sh",
    "modules": [
        {
            "name": "hello",
            "buildsystem": "simple",
            "build-commands": [
                "install -D hello.sh /app/bin/hello.sh"
            ],
            "sources": [
                {
                    "type": "file",
                    "path": "hello.sh"
                }
            ]
        }
    ]
}
```

Now save the file alongside *hello.sh* and call it *org.flatpak.Hello.json*.

In a more complex application, the manifest would list multiple modules. The last one would typically be the application itself, and the earlier ones would be dependencies that are bundled with the app because they are not part of the runtime.

4. Build the application

Now that the app has a manifest, *flatpak-builder* can be used to build it. This is done by specifying the manifest file and a target directory:

```
$ flatpak-builder app-dir org.flatpak.Hello.json
```

This command will build each module that is listed in the manifest and install it to the */app* subdirectory, inside the *app-dir* directory.

5. Test the build

To verify that the build was successful, run the following:

```
$ flatpak-builder --run app-dir org.flatpak.Hello.json hello.sh
```

Congratulations, you've made an app!

6. Put the app in a repository

Before you can install and run the app, it first needs to be put in a repository. This is done by passing the *–repo* argument to *flatpak-builder*:

```
$ flatpak-builder --repo=repo --force-clean app-dir org.flatpak.Hello.json
```

This does the build again, and at the end exports the result to a local directory called *repo*. Note that *flatpak-builder* keeps a cache of previous builds in the *.flatpak-builder* subdirectory, so doing a second build like this is very fast.

This second time we passed in *–force-clean*, which means that the previously created *app-dir* directory was deleted before the new build was started.

7. Install the app

Now we're ready to add the repository that was just created and install the app. This is done with two commands:

```
$ flatpak --user remote-add --no-gpg-verify tutorial-repo repo
$ flatpak --user install tutorial-repo org.flatpak.Hello
```

The first command adds the repository that was created in the previous step. The second command installs the app from the repository.

Both these commands use the *–user* argument, which means that the repository and the app are added per-user rather than system-wide. This is useful for testing.

Note that the repository was added with *–no-gpg-verify*, since a GPG key wasn't specified when the app was built. This is fine for testing, but for official repositories you should sign them with a private GPG key.

8. Run the app

All that's left is to try the app. This can be done with the following command:

```
$ flatpak run org.flatpak.Hello
```

This runs the app, so that it prints *Hello world, from a sandbox*.

1.3 Building

This section includes documentation on how to build apps with Flatpak. It covers runtimes and SDKs in more detail, followed by details on how use `flatpak-builder`.

1.3.1 Runtimes and SDKs

Each Flatpak app is required to have a runtime. This provides the environment that the app runs in, and includes a set of libraries that it can access. The app's runtime must be present on a system for it to run.

Each runtime is paired with an SDK (Software Develpment Kit). For example, for the Freedesktop 1.6 runtime, there is also a Freedesktop 1.6 SDK. The SDK includes the same things as the regular runtime, but also includes all the additional development resources and tools that are required to build an app, such as build and packaging tools, header files, compilers and debuggers.

Applications must be built with the SDK that corresponds to their runtime. For example, an application that uses the Freedesktop 1.6 runtime in order to run must be built with the Freedesktop 1.6 SDK.

Choosing a runtime

When you come to build a Flatpak app, you will need to decide which runtime it will use. An overview of the runtimes that are available can be found in the *Available Runtimes* page. There are deliberately only a small number of runtimes to choose from.

Runtimes require regular maintenance, and application developers should generally not consider creating their own.

Installing a runtime and SDK

Once you have chosen a runtime for your application, it is necessary to install it as well as the matching SDK. (Runtimes and SDKs are installed in exactly the same way.)

For example, the command to install the GNOME 3.25 runtime and SDK is:

```
$ flatpak install flathub org.gnome.Platform//3.26 org.gnome.Sdk//3.26
```

A number of the examples in these docs use this runtime and SDK, so it is a good idea to try this command yourself.

1.3.2 Flatpak Builder

Flatpaks are built using the `flatpak-builder` tool. This allows a series of modules to be built into a single application bundle. These modules can include libraries and dependencies in addition to the application itself.

Modules can be built with a variety of build systems, including autotools, cmake, cmake-ninja, meson, and the so called Build API. A "simple" build method is also available, which allows a series of commands to be specified.

Exporting

The result of the build process can be exported to a repository or automatically installed locally.

Exporting to a repository

The `--repo` option allows a repository to be specified, which the resulting application will be added to. This takes the format:

```
$ flatpak-builder --repo=<repository-destination> application.id.json
```

Note: By default, flatpak-builder splits off translations and debug information into separate *.Locale* and *.Debug* extensions. These extensions are automatically exported into a repository along with the application.

Installing builds directly

Instead of exporting to a repository, the application bundle that is produced by `flatpak-builder` can be automatically installed locally:

```
$ flatpak-builder --install application.id.json
```

Signing

Every commit to a Flatpak repository should be signed with a GPG signature. If `flatpak-builder` is being used to modify or create a repository, a GPG key should therefore be passed to it. This can be done with the `--gpg-sign` option, such as:

```
$ flatpak-builder --gpg-sign=<key-id> --repo=<repository-destination> application.id.
↪json
```

The `--gpg-homedir` option can also be used to specify the home directory of the key that is being used.

Though it generally isn't recommended, it is possible not to use GPG verification. In this case, the `--no-gpg-verify` option should be used when adding the repository. Note that it is necessary to become root in order to update a repository that does not have GPG verification enabled.

1.3.3 Manifests

The input to `flatpak-builder` is a JSON file that describes the parameters for building an application, as well as instructions for each of the modules that are to be built. This file is called the manifest.

The following example is the manifest for the GNOME Dictionary application. It is short, because only one module is built - the application itself:

```
{
  "app-id": "org.gnome.Dictionary",
  "runtime": "org.gnome.Platform",
  "runtime-version": "3.26",
  "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/3.26/gnome-
↪dictionary-3.26.0.tar.xz",
          "sha256": "387ff8fbb8091448453fd26dcf0b10053601c662e59581097bc0b54ced52e9ef"
```

```
            }
        ]
    }
  ]
}
```

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. Module sources can be of several types, including `.tar` or `.zip` archives, Git or Bzr repositories, patch files or shell commands that are run.

Each of the properties that can be specified in a manifest file are listed in the *Flatpak Builder Command Reference*, as well as the `flatpak-builder` man page.

finish-args

Flatpaks have extremely limited access to the host environment by default. However, most applications require access to resources outside of their sandbox in order to be useful. This can be achieved with the `finish-args` manifest section, which allows sandbox permissions to be configured.

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.

A list of `finish-args` options can be found in *Sandbox Permissions*.

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-file` properties.

Example manifests

A complete manifest for GNOME Dictionary built from Git. It is also possible to browse all the manifests hosted by Flathub.

1.3.4 Tutorial

This tutorial provides a sample set of step that you can use to try `flatpak-builder` yourself. In it, you will learn how to use `flatpak-builder` to build the GNOME Dictionary applicaiton.

1. Create a manifest

To create a manifest for the application, create a file called `org.gnome.Dictionary.json` and paste the Dictionary manifest JSON from *Manifests* into it.

2. Run the build

To use the manifest to build the Dictionary application, run the following command:

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

This will:

- Create a new directory called dictionary
- 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 X11 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`).

3. Add the new repository

To test the application that has been built, you need to add the new repository that has been created:

```
$ flatpak --user remote-add --no-gpg-verify --if-not-exists tutorial-repo tutorial-
↪repo
```

4. Install the application

The next step is to install the Dictionary application from the repository. To do this, run:

```
$ flatpak --user install tutorial-repo org.gnome.Dictionary
```

To check that the application has been successfully installed, 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
```

5. Run the application

Finally, you can run the application that you've built:

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

1.4 Publishing

Flatpak provides several ways to distribute applications to users. For many applications, the most convenient and effective method is to use Flathub, which provides a large centralized repository of Flatpak applications.

Alternatively, it is possible to host a repository yourself, or to distribute Flatpaks as single file bundles.

1.4.1 Repositories

Flatpak repositories are the primary mechanism for publishing applications, so that they can be installed by users.

Some aspects of repositories are addressed by other sections of the documentation. Basic commands for adding, removing and inspecting repositories can be found in the *Using Flatpak* section. Additionally, the section on *Flatpak Builder* covers the most common method for adding applications to repositories.

To use a repository to publish an application, it is possible to either host your own (covered in the next section, *Hosting a repository*) or use Flathub, the primary publishing and hosting service for Flatpak applications.

Software center applications like GNOME Software or KDE Discover allow browsing repositories, and can also dynamically promote new or popular applications. If you use Flathub, the repository will typically have already been added by users, so adding an application to the repository is sufficient to make it available to them.

.flatpakref files

`.flatpakref` files can be used in combination with repositories to provide an additional, easy way for users to install an application, often by clicking on the file or a download link.

Internally, `.flatpakref` files are simple description files that include information about a Flatpak application. An example:

```
[Flatpak Ref]
Name=fr.free.Homebank
Branch=stable
Title=fr.free.Homebank from flathub
Url=https://dl.flathub.org/repo/
RuntimeRepo=https://dl.flathub.org/repo/flathub.flatpakrepo
IsRuntime=false
GPGKey=mQINBFlD2sABEADsiUZUO...
```

As can be seen, the file includes the ID of the application and the location of the repository that contains it, as well a link to information about the repository that provides the application's runtime. `.flatpakref` files therefore contain all the information needed to install an application.

Note: `.flatpakref` files should include the base64-encoded version of the GPG key that was used to sign the repository. This can be obtained with the following command:

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

One advantage of `.flatpakref` files is that they can be used to install applications even if their repository hasn't been added by the user. In this case the repository that contains the application will either be automatically installed, or the user will be prompted to install it. This will also happen if the necessary runtime isn't present.

`.flatpakref` can be used to install applications from the command line as well as with graphical software installers. This is done with the standard `flatpak install` command, which accepts both local and remote `.flatpakref` files. For example:

```
$ flatpak install https://flathub.org/repo/appstream/fr.free.Homebank.flatpakref
```

Or, if the same file has been downloaded:

```
$ flatpak install fr.free.Homebank.flatpakref
```

Publishing updates

Flatpak repositories are similar a Git repositories, in that they store every version of an application by keeping a record of the difference between each version. This makes updating efficient, since only the difference (or "delta") between two versions needs to be downloaded when an update is performed.

When a new version of an application is added to a repository, it immediately becomes available to users. Software centers are able to automatically check for and install new versions. Those who are using the command line have to manually run `flatpak update` to check for and install new versions of any applications they have installed.

1.4.2 Hosting a repository

The section on *Flatpak Builder* describes how to generate repositories. 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 involve a lot of HTTP requests. Since new requests can be slow, it is important to enable HTTP keep-alive on the web server that is hosting your repository.

OSTree supports something called static deltas. These are single files that contain 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.

.flatpakrepo files

`.flatpakrepo` files are a convenient way to let users add a repository. These are simple description files which contain information about the repository. For example, the Flathub repo file looks like:

```
[Flatpak Repo]
Title=Flathub
Url=https://dl.flathub.org/repo/
Homepage=https://flathub.org/
```

```
Comment=Central repository of Flatpak applications
Description=Central repository of Flatpak applications
Icon=https://dl.flathub.org/repo/logo.svg
GPGKey=mQINBFlD2sABEADsiUZUO...
```

Here you can see that the repo file contains descriptive metadata, such as the repository name, description, icon and website. The file also contains information that is needed to add the repository, including a download URL and the repository's GPG key.

`.flatpakrepo` files can be used to add a repository from the command line. For example, the command to add Flathub using its repo file is:

```
$ flatpak remote-add --if-not-exists flathub https://dl.flathub.org/repo/flathub.
↪flatpakrepo
```

The command line isn't the only way to add a repository using a `.flatpakrepo` file - on desktops that support Flatpak, it is just a matter of clicking the repository file or a download link that points to it.

Note: `.flatpakrepo` files should include the base64-encoded version of the GPG key that was used to sign the repository. This can be obtained with the following command:

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

1.4.3 Single-file bundles

Hosting a repository is the preferred way to distribute an application, since repositories allow applications to be updated. However, sometimes it can be appropriate to use a single-file bundle. These can be used to provide a direct download of the application, to distribute applications using removable media, or to send them as email attachments.

Flatpak allows single file bundles to be created with the `build-bundle` and `build-import-bundle` commands, which allow an application in a repository to be converted into a bundle and back again:

```
$ 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
```

1.5 Reference Documentation

Reference documentation for flatpak and flatpak-builder.

1.5.1 Flatpak Command Reference

1.5.2 Flatpak Builder Command Reference

1.5.3 Available Runtimes

This page provides information about available Flatpak runtimes. It is primarily intended as information for application developers and distributors.

There are currently three main runtimes available: Freedesktop, GNOME and KDE. These are all hosted on Flathub.

Freedesktop

The Freedesktop runtime is the standard runtime that can be used for any application and contains a set of essential libraries and services, including D-Bus, GLib, PulseAudio, X11 and Wayland.

Available Freedesktop runtimes:

ID	Description
org.freedesktop.Platform	Runtime
org.freedesktop.Platform.Locale	Runtime translations (extension)
org.freedesktop.Sdk	SDK
org.freedesktop.Sdk.Debug	SDK debug information (extension)
org.freedesktop.Sdk.Locale	SDK translations (extension)
org.freedesktop.Sdk.Docs	SDK documentation (extension)

GNOME

The GNOME runtime is appropriate for any application that uses the GNOME platform. It is based on the Freedesktop runtime and adds the GNOME platform, including:

- Clutter
- Gjs
- GObject Introspection
- GStreamer
- GVFS
- Libnotify
- Libsecret
- LibSoup
- PyGObject
- Vala
- WebKitGTK

Available GNOME runtimes:

ID	Description
org.gnome.Platform	Runtime
org.gnome.Platform.Locale	Runtime translations (extension)
org.gnome.Sdk	SDK
org.gnome.Sdk.Debug	SDK debug information (extension)
org.gnome.Sdk.Locale	SDK translations (extension)
org.gnome.Sdk.Docs	SDK documentation (extension)

KDE

The KDE runtime is also based on the Freedesktop runtime and adds Qt and KDE Frameworks. It is appropriate for any application that makes use of the KDE platform and most Qt-based applications.

Available KDE runtimes:

ID	Description
org.kde.Platform	Runtime
org.kde.Platform.Locale	Runtime translations (extension)
org.kde.Sdk	SDK
org.kde.Sdk.Debug	SDK debug information (extension)
org.kde.Sdk.Locale	SDK translations (extension)
org.kde.Sdk.Docs	SDK documentation (extension)

1.5.4 Sandbox Permissions

Sandbox permissions can be configured from an application manifest (see *Manifests*). They can also be set with the `build-finish`, `run` and `override` commands.

Permission options

The following list includes some of the most useful permission options. The full list can be viewed using `flatpak build-finish --help`

`--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

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
```

[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.