Flatpak Documentation

Flatpak Team

Oct 24, 2019

Contents

These docs cover everything you need to know to build and distribute applications using Flatpak. They begin with a basic introduction to Flatpak, background information on basic concepts, and a guide to the Flatpak command line interface. Later sections provide detailed information on building and distributing 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 use Flatpak to install and run applications, please refer to the Flatpak website.

CHAPTER 1

Contents

1.1 Introduction to Flatpak

Flatpak is a framework for distributing desktop applications on Linux. It has been created by developers who have a long history of working on the Linux desktop, and is run as an independent open source project.

1.1.1 Target audience

Flatpak can be used by all kinds of desktop applications, and aims to be as agnostic as possible regarding how applications are built. There are no requirements regarding which programming languages, build tools, toolkits or frameworks can be used.

While Flatpak only runs on Linux, it can be used by applications that target other operating systems, as well as those that are Linux-specific. Applications can be open source or proprietary (although some distribution services, like Flathub, can have restrictions in this respect).

The only technical requirements made by Flatpak are that applications follow a small number of Freedesktop standards, in order to enable desktop integration (see *Requirements & Conventions*).

1.1.2 Reasons to use Flatpak

Flatpak has some major advantages over other approaches to distributing applications on Linux. First and foremost, Flatpak allows a single application build to be installed and run on virtually any Linux distribution. It can also be used in combination with Flathub, a centralized service for distributing applications on all distributions. This makes it possible for application developers to target the entire Linux desktop market from one place.

Flatpak also offers native integration for the main Linux desktops, so that users can easily browse, install, run and use Flatpak applications through their existing desktop environment and tools.

Other benefits for developers include:

- **Forward-compatibility:** the same Flatpak can be run on different versions of the same distribution, including versions that haven't been released yet. This doesn't require any changes or management by application developers.
- **Maintained platforms:** called runtimes, these contain collections of dependencies, which can be used by applications, and which can take a lot of the work out of application development.
- **Bundling:** this allows application developers to ship almost any dependency or library as part of their application. This gives complete control over which software is used to build applications.
- **Consistent application environments:** because these are the same across devices, applications perform as intended. This also makes it easier to identify bugs and to do testing.

Finally, while Flatpak does provide a centralized service for distributing applications, it also allows decentralized hosting and distribution, so that application developers or downstreams can host their own applications and application repositories.

Information about Flatpak's internals can be found in *Under the Hood*.

1.2 Getting Started

This section includes an introduction to basic Flatpak concepts, a guide on how to use the Flatpak command line interface, and a tutorial for building a simple application.

To complete this section, Flatpak should have been installed and the Flathub repository should have been enabled. The Flatpak website provides instructions for how to do this with a range of distributions.

1.2.1 Basic concepts

Flatpak can be understood through a small number of key concepts. It is useful to be familiar with these before learning about how to use Flatpak from the command line, or using it to build applications.

Runtimes

Runtimes provide the basic dependencies that are used by applications. 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 (Flatpak can automatically install the runtime required by an application). Multiple different runtimes can be installed at the same time, including different versions of the same runtime.

Runtimes are distribution agnostic and do not depend on particular distribution versions. This means that they provide a stable, cross-distribution base for applications, and allow applications to continue to work irrespective of operating system updates.

Bundled libraries

If an application requires any dependencies that aren't in its runtime, they can be bundled as part of the application. This gives application developers flexibility regarding the dependencies that they use, including using:

- libraries that aren't available in a runtime
- different versions of libraries from the ones that are in a runtime
- patched versions of libraries

Sandboxes

With Flatpak, each application is built and run in an isolated environment, which is called the 'sandbox'. Each sandbox contains an application and its runtime. By default, the application can only access the contents of its sandbox. Access to user files, network, graphics sockets, subsystems on the bus and devices have to be explicitly granted. 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. They give the ability to interact with data, files and services without the need to add sandbox permissions.

Examples of capabilities that can be accessed through portals include opening files through a file chooser dialog, or printing. Interface toolkits can implement transparent support for portals, so access to resources outside of the sandbox will work securely and out of the box.

More information about portals can be found in *Sandbox Permissions*.

Repositories

Flatpak applications and runtimes are typically stored and published using repositories, which behave very similarly to Git repositories. A Flatpak repository can contain a single object or multiple objects, and each object is versioned, which allows upgrading and even downgrading.

Each system which is using Flatpak can be configured to access any number of remote repositories. Once a system has been configured to access a 'remote', the remote repository's content can be inspected and searched, and it can be used as the source of applications and runtimes.

When an update is performed, new versions of installed applications and runtimes are downloaded from the relevant remotes. Like with Git, only the difference between versions is downloaded, which makes the process very efficient.

1.2.2 Using Flatpak

This page provides an introduction to the `flatpak` command line interface, and explains essential technical conventions as well as the most common commands.

End users shouldn't generally need to use this page or the Flatpak command line interface, since Flatpak can be easily used through graphical software management tools, though they are of course free to use the command line if they prefer!

The flatpak command

`flatpak` is the primary Flatpak command, to which specific commands are appended. For example, the command to install something is `flatpak install` and the command to uninstall is `flatpak uninstall`.

Identifiers

Flatpak identifies each application and runtime using a unique three-part identifier, such as `com.company.App`. The final segment of this address is the object's name, and the preceding part identifies the developer, so that the same developer can have multiple applications, like `com.company.App1` and `com.company.App2`.

Identifier triples

Typically it is sufficient to refer to objects using their ID. However, in some situations it is necessary to refer to a specific version of an object, or to a specific architecture. For example, some applications might be available as a stable and a testing version, in which case it is necessary to specify which one you want to install.

Flatpak allows architectures and versions to be specified using an object's identifier triple. This takes the form of `name/architecture/branch`, such as `com.company.App/i386/stable`. (Branch is the term used to refer to versions of the same object.) The first part of the triple is the ID, the second part is the architecture, and the third part is the branch.

Identifier triples can also be used to specify just the architecture or the branch, by leaving part of the triple blank. For example, `com.company.App//stable` would just specify the branch, and `com.company.App/i386//` just specifies the architecture.

The Flatpak CLI provides feedback if the architecture or branch of an object needs to be specified.

System versus user

Flatpak commands can be run either system-wide or per-user. Applications and runtimes that are installed system-wide are available to all users on the system. Applications and runtimes that are installed per-user are only available to the users that installed them.

The same principle applies to repositories - repositories that have been added system-wide are available to all users, whereas per-user repositories can only be used by a particular user.

Flatpak commands are run system-wide by default. If you are installing applications for day-to-day usage, it is recommended to stick with this default behavior.

However, running commands per-user can be useful for testing and development purposes, since objects that are installed in this way won't be available to other users on the system. To do this, use the `--user` option, which can be used in combination with most `flatpak` commands.

Commands behave in exactly the same way if they are run per-user rather than system-wide.

Basic commands

This section covers basic commands needed to install, run and manage Flatpak applications. For the full list of Flatpak commands, run `flatpak --help` or see the *Flatpak Command Reference*.

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

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.

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`.

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.

Since Flatpak 1.2, the `install` command can search for applications. A simple:

```
$ flatpak install gimp
```

will confirm the remote and application and proceed to install.

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
```

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
```

Troubleshooting

Flatpak has a few commands that can help you to get things working again when something goes wrong.

To remove runtimes and extensions that are not used by installed applications, use:

```
$ flatpak uninstall --unused
```

To fix inconsistencies with your local installation, use:

```
$ flatpak repair
```

Flatpak also has a number of commands to manage the portal permissions of installed apps. To reset all portal permissions for an app, use `flatpak permission-reset`:

```
$ flatpak permission-reset org.gimp.GIMP
```

To find out what changes have been made to your Flatpak installation over time, you can take a look at the logs (since 1.2):

```
$ flatpak history
```

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.

In order to complete this tutorial, you should have followed the setup guide on flatpak.org. You also need to have installed `flatpak-builder`, which is usually available from the same repository as the `flatpak` package (e.g. use `apt` or `dnf`). You can also install it as a flatpak with `flatpak install flathub org.flatpak.Builder`.

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 18.08 runtime and SDK. To install these, run:

```
$ flatpak install flathub org.freedesktop.Platform//18.08 org.freedesktop.Sdk//18.08
```

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": "18.08",
    "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 build-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 `build-dir` directory.

5. Test the build

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

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

Congratulations, you've made an app!

Using `--run` results in a sandbox with mostly the same permissions as the final app, with the exception of filesystem permissions. As such it shouldn't be relied upon beyond basic testing.

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 build-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 `build-dir` directory was deleted before the new build was started.

In order for your application to show up in application stores while testing with a local repository, you might have to run `flatpak build-update-repo repo`. For more information how to publish to application stores see Appdata files.

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 contains detailed information on how to build applications as Flatpaks. It starts with an overview of the build process, before diving into requirements for applications, guidance on key decisions, information on how to use `flatpak-builder`, and how to write manifest files.

If you haven't already, it is recommended to run through *Building your first Flatpak* before reading this section.

1.3.1 Building Introduction

Building your first Flatpak has already provided a quick demonstration of how applications get built with Flatpak. This page provides an additional general overview of what's involved.

flatpak-builder

`flatpak-builder` is the primary tool for building Flatpak applications. It allows you to take the source files for an application and build it into a Flatpak application. It also allows multiple other dependencies to be built at the same time, which get bundled into the build.

The input to `flatpak-builder` is a manifest file. This specifies the parameters for the application that will be built, such as its name and which runtime it will depend on. The manifest also lists all the modules that are to be built as part of the build process. A source for each module can be specified, including links to file archives or version control repositories. One of the modules (usually the last one) is the application code itself.

The basic format used to invoke `flatpak-builder` is:

```
$ flatpak-builder <build-dir> <manifest>
```

Where `<build-dir>` is the path to the directory that the application will be built into, and `<manifest>` is the path to a manifest file. The contents of `<build-dir>` can be useful for testing and debugging purposes, but is generally treated as an artifact of the build process.

When `flatpak-builder` is run:

- The build directory is created, if it doesn't already exist
- The source code for each module is downloaded and verified

- The source code for each module is built and installed
- The build is finished, by setting sandbox permissions
- The build result is exported to a repository (which will be created if it

doesn't exist already)

The application can then be installed from the repository and run.

Software Development Kits (SDKs)

Instead of being built using the host environment, Flatpak applications are built inside a separate environment, called an SDK.

SDKs are like the regular runtime that applications run in. The difference is that SDKs also include all the development resources and tools that are required to build an application, such as build and packaging tools, header files, compilers and debuggers.

Each runtime has an accompanying SDK. For example, there is both a GNOME 3.26 runtime and a GNOME 3.26 SDK. Applications that use the runtime are built with the matching SDK.

Like runtimes, SDKs will sometimes be automatically installed for you, but if you do need to manually install them, they are installed in the same way as applications and runtimes, such as:

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

1.3.2 Requirements & Conventions

Flatpak deliberately makes as few requirements of applications as possible. However, a small number of standard Linux desktop conventions are expected, primarily to ensure that applications integrate with Linux desktops and app centers. Developers might also encounter a small number of Linux technical conventions.

Information on further desktop integration options can be found in *Desktop Integration*.

Expected Standards

Applications that use Flatpak are generally expected to comply with the following standards. Applications that have previously targeted the Linux desktop will typically need to make very few (if any) changes to do this.

Application IDs

As described in *Using Flatpak*, Flatpak requires each application to have a unique identifier, which has a form such as `org.gnome.Dictionary`. The format is in reverse-DNS style so the first section is a domain controlled by the project and the trailing section represents the specific project. As will be seen below and in future sections, this ID is expected to be used in a number of places. Developers must follow the standard D-Bus naming conventions when creating their own IDs. This format is already recommended by the Desktop File specification and Appstream specification also.

For some practical examples of bad IDs

- `org.example.desktop`

  This is a bad ID because the Appstream standard for legacy reasons treats IDs ending with `.desktop` as a special case causing inconsistency. For this same reason, `.Desktop` suffixes should not be used for newly

named applications. Don't hesitate to repeat the application name even if it already is part of the domain name section of the identifier (eg. `org.example.Example`).

- `io.github.Foo`

  This is problematic because while `foo.github.io` may be unique to your project it does not include a project specific identifier. This may cause issues if another project creates `io.github.Foo-Bar` which should be its own namespace but areas of `flatpak` may treat them similar. A better ID would be `io.github.foo.Foo` even if its redundant.

- `org.example-site.Foo`

  This ID is not valid according to the DBus specification. You can use `org.example_site.Foo` instead.

- `com.github.foo.Bar`

  While a project may be hosted on GitHub it does not have any control over the `github.com` domain. Instead you should use `io.github` as shown above.

AppData files

AppData files provide metadata about applications, which is used by application stores (such as Flathub, GNOME Software and KDE Discover). The Freedesktop AppStream specification provides a complete reference for providing AppData.

AppData files should be named with the application ID and the `.appdata.xml` file extension, and should be placed in `/app/share/metainfo/`. For example:

```
/app/share/metainfo/org.gnome.Dictionary.appdata.xml
```

The `appstream-util validate-relax` command can be used to check AppData files for errors.

Application icons

Applications are expected to provide an application icon, which is used for their application launcher. These icons should be provided in accordance with the Freedesktop icon specification.

Icons should be named with the application's ID, be in either PNG or SVG format, and must be placed in the standard location:

```
/app/share/icons/hicolor/$size/apps/
```

For example, the path to the 128128px version of GNOME Dictionary's icon is:

```
/app/share/icons/hicolor/128x128/apps/org.gnome.Dictionary.png
```

Desktop files

Desktop files are used to provide the desktop environment with information about each application. The Freedesktop specification provides a complete reference for writing desktop files, and additional information about them is available online.

Desktop files should be named with the application's ID, followed by the `.desktop` file extension, and should be placed in `/app/share/applications/`. For example:

```
/app/share/applications/org.gnome.Dictionary.desktop
```

A minimal desktop file should contain at least the application's *name*, *exec* command, *type*, *icon* name and *categories*:

```
[Desktop Entry]
Name=Gnome Dictionary
Exec=org.gnome.Dictionary
Type=Application
Icon=org.gnome.Dictionary
Categories=GNOME;GTK;Office;Dictionary;
```

The `desktop-file-validate` command can be used to check for errors in desktop files.

Exporting through extra-data

Files downloaded through `extra-data` are only downloaded when installing, as such they aren't yet available for `flatpak-builder` to automatically export during the build process.

When using `extra-data`, place any files that must be exported under this location:

```
/app/extra/export/share/
```

For example, if GNOME Dictionary used `extra-data` to download a 96x96 icon this would be its path:

```
/app/extra/export/share/icons/hicolor/96x96/apps/org.gnome.Dictionary.png
```

Technical conventions

The following are standard technical conventions used by Flatpak and Linux desktops. Those with Linux experience will likely already be aware of them. However, developers who are new to Linux might find some of this information useful.

D-Bus

D-Bus is the standard IPC framework used on Linux desktops. A lot of applications won't need to use it, but it is supported by Flatpak should it be required.

D-Bus can be used for application launching and communicating with some system services. Applications can also provide their own D-Bus services (when doing this, the D-Bus service name is expected to be the same as the application ID).

Filesystem layout

Each Flatpak sandbox, which is the environment in which an application is run, contains the filesystem of the application's runtime. This follows standard Linux filesystem conventions.

For example, the root of the sandbox contains the `/etc` directory for configuration files and `/usr` for multi-user utilities and applications. In addition to this, each sandbox contains a top-level `/app` directory, which is where the application's own files are located.

XDG base directories

XDG base directories are standard locations for user-specific application data. Popular toolkits provide convenience functions for accessing XDG base directories. These include:

- Electron: XDG base directories can be accessed with `app.getPath`
- Glib: provides access to the XDG base directories through the `g_get_user_cache_dir ()`, `g_get_user_data_dir ()`, `g_get_user_config_dir ()` functions
- Qt: provides access to XDG base directories with the QStandardPaths Class

However, applications that aren't using one of these toolkits can expect to find their XDG base directories in the following locations:

Base directory	Usage	Default location
XDG_CONFIG_HOME	User-specific configuration files	~/.var/app/<app-id>/config
XDG_DATA_HOME	User-specific data	~/.var/app/<app-id>/data
XDG_CACHE_HOME	Non-essential user-specific data	~/.var/app/<app-id>/cache

For example, GNOME Dictionary will store user-specific data in:

```
~/.var/app/org.gnome.Dictionary/data/gnome-dictionary
```

Note that applications can be configured to use non-default base directory locations (see *Sandbox Permissions*).

1.3.3 Dependencies

Flatpak provides a number of different options for how applications can depend on other software. When setting out to build an application with Flatpak for the first time, it is therefore necessary to decide how application dependencies will be organized.

This page outlines what the options are, and provides guidance on when to use each one.

Runtimes

As was described in *Basic concepts*, runtimes provide basic dependencies that can be used by applications. They also provide the environment that applications run in. Flatpak requires each application to specify a runtime. Therefore, one of the first decisions you need to make when building an application with Flatpak, is 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. Typically, runtimes are picked on the basis of which dependencies an application requires. If a runtime exists that provides libraries that you plan on using, this is usually the correct runtime to use!

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

Runtimes are automatically installed for users when they install an application, and build tools can also automatically install them for you (`flatpak-builder`'s `--install-deps-from` option is useful for this). However, if you do need to manually install your chosen runtime, this can be done in the same way as installing an application, with the `flatpak install` command. For example, the command to install the GNOME 3.26 runtime is:

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

Bundling

One of the key advantages of Flatpak is that it allows application authors to bundle whatever libraries or dependencies that they want. This means that developers aren't constrained by which libraries are available through Linux distributions.

When it comes to building an application for the first time, you will need to decide which dependencies to bundle. This can include:

- libraries that aren't in your chosen runtime
- different versions of libraries that are in your chosen runtime
- patched versions of libraries
- data or other resources that form part of the application

As will be seen, bundled dependencies can be automatically downloaded as part of the build process. It is also possible to apply patches and perform other transformations.

While bundling is very powerful and flexible, it also places a greater maintenance burden on the application developer. Therefore, while it is possible to bundle as much as you would like, it is generally recommended to try and keep the number of bundled modules as low as possible. If a dependency is available as part of a runtime, it is generally better to use that version rather than bundle it yourself.

The specifics of how to bundle libraries is covered in the *Manifests* section.

Base apps

Runtimes and bundling are the two main ways that dependencies are handled with Flatpak. They allow applications to rely on stable collections of dependencies on the one hand, and to have flexibility and control on the other.

However, in some cases, dependencies come as part of a bigger framework or toolkit, which doesn't fit into a runtime but which is also cumbersome to manually bundle as a series of individual modules. This is where *base apps* come in.

Base apps contain collections of bundled dependencies which can then be bundled as part of an application. They don't get rebuilt as part of the build process, which makes building faster (particularly when bundling large dependences). And because each base app is only built once, it is guaranteed to be identical wherever it is used, so it will only be saved once on disk.

Base apps are a relatively specialized concept and only some applications need to use them (the most common base app is used for Electron applications). However, if your application uses a large, complex or specialized framework, it is a good idea to check for available base apps before you start building.

1.3.4 Flatpak Builder

`flatpak-builder` has already been introduced in *Building your first Flatpak* and *Building Introduction*. This page provides additional detail on how to use `flatpak-builder`, including the various command options that are available.

Exporting

`flatpak-builder` provides two options for exporting an application in order to run it. The first is to export to a repository, from which the application can be run. The second is to automatically install locally.

Exporting to a repository

The `--repo` option allows a repository to be specified, for the application to be exported to. This takes the format:

```
$ flatpak-builder --repo=<repo> <build-dir> <manifest>
```

Here, `<repo>` is a path to a repository. If no repository exists at the specified location, the repository will be created. If the application is already in the specified repository, `flatpak-builder` will add the build as a new version of the existing application.

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 Flatpak that is produced by `flatpak-builder` can be automatically installed locally, using the `--install` option:

```
$ flatpak-builder --install <build-dir> <manifest>
```

This approach has the advantage of skipping the separate install step that is needed when exporting to a repository.

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> --repo=<repository> <manifest>
```

Here, `<key>` is the ID of the GPG key that is to be used. 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.

1.3.5 Manifests

The input to `flatpak-builder` is a JSON or YAML 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.

This page provides information and guidance on how to use manifests, including an explanation of the most common parameters that can be specified. It is recommended to have followed the *Building your first Flatpak* tutorial before reading this section, and to be familiar with *Flatpak Builder*.

Manifest files should be named using the application ID. For example, the manifest file for GNOME Dictionary is named `org.gnome.Dictionary.json`. This page uses this manifest file, which was introduced in *Building your first Flatpak*, for all its examples.

A complete list of all the properties that can be specified in manifest files can be found in the *Flatpak Builder Command Reference*, as well as the `flatpak-manifest` man page.

Basic properties

Each manifest file should specify basic information about the application that is to be built, including the `app-id`, `runtime`, `runtime-version`, `sdk` and `command` parameters. These properties are typically specified at the beginning of the file.

For example, the GNOME Dictionary manifest includes:

```
"app-id": "org.gnome.Dictionary",
"runtime": "org.gnome.Platform",
"runtime-version": "3.26",
"sdk": "org.gnome.Sdk",
"command": "gnome-dictionary",
```

Specifying a runtime and runtime version allows that the runtime that is needed by your application to be automatically installed on users' systems.

File renaming

As was described in the *Introduction to Flatpak*, exports are application files that are made available to the host, and include things like the application's `.desktop` file and icon.

The names of files that are exported by a Flatpak must prefixed using the application ID, such as `org.gnome.Dictionary.desktop`. The best way to do this is to rename these files directly in the application's source.

If renaming exported files to use the application ID is not possible, `flatpak-builder` allows them to be renamed as part of the build process. This can be done by specifying one of the following properties in the manifest:

- `rename-icon` - rename the application icon
- `rename-desktop-file` - rename the `.desktop` filename
- `rename-appdata-file` - rename the AppData file

Each of these properties accepts the name of the source file to be renamed. `flatpak-builder` then automatically renames the file to match the application ID. Note that this renaming method can introduce internal naming conflicts, and that renaming files in tree is therefore the most reliable approach.

Finishing

Applications that are run with Flatpak have extremely limited access to the host environment by default, but applications require access to resources outside of their sandbox in order to be useful. Finishing is the build stage where the application's sandbox permissions are specified, in order to give access to these resources.

The finishing manifest section uses the `finish-args` property, which can be seen in the Dictionary manifest file:

```
"finish-args": [
   "--socket=x11",
   "--share=network"
],
```

As was explained in *Building your first Flatpak*, these two finishing properties give the application access to the X11 display server and to the network. Guidance on which permissions to use can be found in *Sandbox Permissions*, and a full list of `finish-args` options can be found in *Sandbox Permissions*.

Cleanup

The cleanup property can be used to remove files produced by the build process that are not wanted as part of the application, such as headers or developer documentation. 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.

Modules

The module list specifies each of the modules that are to be built as part of the build process. One of these modules is the application itself, and other modules are dependencies and libraries that are bundled as part of the Flatpak. While simple applications may only specify one or two modules, and therefore have short modules sections, some applications can bundle numerous modules and therefore have lengthy modules sections.

GNOME Dictionary's modules section is short, since it just contains the application itself, and looks like:

```
"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, each listed module has a `name` (which can be freely assigned) and a list of `sources`. Each source has a `type`, and available types include:

- `archive` - `.tar` or `.zip` archive files
- `git` - Git repositories
- `bzr` - Bazaar repositories
- `file` - local file (these are copied into the source directory)
- `dir` - local directory (these are copied into the source directory)
- `script` - an array of shell commands (these are put in a shellscript file)
- `shell` - an array of shell commands that are run during source extraction
- `patch` - a patch (are applied to the source directory)
- `extra-data` - data that can be downloaded at install time; this can include archive or package files

Different properties are available for each source type, which are listed in the *Flatpak Builder Command Reference*.

Supported build systems

Modules can be built with a variety of build systems, including:

- autotools
- cmake
- cmake-ninja
- meson
- the "Build API"

A "simple" build method is also available, which allows a series of commands to be specified.

Example manifests

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

1.3.6 Sandbox Permissions

One of Flatpak's main goals is to increase the security of desktop systems by isolating applications from one another. This is achieved using sandboxing and means that, by default, applications that are run with Flatpak have 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 X11, system D-Bus, or PulseAudio.

Most applications will need access to some of these resources in order to be useful. This is primarily done during the finishing build stage, which can be configured through the `finish-args` section of the manifest file (see *Manifests*).

Portals

Portals have already been mentioned in the *Introduction to Flatpak*. They are a framework for providing access to resources outside of the sandbox, including:

- Opening files with a native file chooser dialog
- Opening URIs
- Printing
- Showing notifications
- Taking screenshots
- Inhibiting the user session from ending, suspending, idling or getting switched away

- Getting network status information

In many cases, portals use a system component to implicitly ask the user for permission before granting access to a particular resource. For example, in the case of opening a file, the user's selection of a file using the file chooser dialog is interpreted as implicitly granting the application access to whatever file is chosen.

This approach enables applications to avoid having to configure blanket access to large amounts of data or services and gives users control over what their applications have access to.

Interface toolkits like GTK3 and Qt5 implement transparent support for portals, meaning that applications don't need to do any additional work to use them (it is worth checking which portals each toolkit supports). 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 use them.

Permissions guidelines

While application developers have control over the sandbox permissions they wish to configure, good practice is encouraged and can be enforced. For example, the Flathub hosting service places requirements on which permissions can be used, and software on the host may warn users if certain permissions are used.

The following guidelines describe which permissions can be freely used, which can be used on an as-needed basis, and which should be avoided.

Standard permissions

The following permissions provide access to basic resources that applications commonly require, and can therefore be freely used:

- `--share=network` - access the network
- `--socket=x11` - show windows using X11
- `--socket=fallback-x11` - show windows using X11, if Wayland is not available
- `--share=ipc` - share IPC namespace with the host (necessary for X11)
- `--socket=wayland` - show windows with Wayland
- `--device=dri` - OpenGL rendering
- `--socket=pulseaudio` - play sound with PulseAudio

D-Bus access

Access to the entire bus with `--socket=system-bus` or `--socket=session-bus` should be avoided, unless the application is a development tool.

Ownership

Applications are automatically granted access to their own namespace. Ownership beyond this is typically unnecessary, although there are a small number of exceptions, such as using MPRIS to provide media controls.

Talk

Talk permissions can be freely used, although it is recommended to use the minimum required.

Filesystem access

It is common for applications to require access to different parts of the host filesystem, and Flatpak provides a flexible set of options for this. Some examples include:

- `--filesystem=host` - access normal files on the host, not including host os or system internals described below
- `--filesystem=home` - access the user's home directory
- `--filesystem=/path/path` - access specific paths
- `--filesystem=xdg-download` - access a specific XDG folder

As a general rule, Filesystem access should be limited as much as possible. This includes using:

- Using portals as an alternative to blanket filesystem access, wherever

possible. - Using read-only access wherever possible, using the `:ro` option. - If some home directory access is absolutely required, using XDG directory access only.

The full list the available filesystem options can be found in the *Sandbox Permissions*. Other filesystem access guidelines include:

- The `--persist=path` option can be used to map paths from the user's home directory into the sandbox filesystem. This makes it possible to avoid configuring access to the entire home directory, and can be useful for applications that hardcode file paths in `~/`.
- If an application uses `$TMPDIR` to contain lock files you may want to add `--env=TMPDIR=/var/tmp` or if it uses `$TMPDIR` to share with processes outside the sandbox you will want a wrapper script that sets it to `$XDG_CACHE_HOME`.
- Retaining and sharing configuration with non-Flatpak installations is to be avoided.

As mentioned above the `host` option does not actually provide complete access to the host filesystem. The main rules are:

- These directories are blacklisted: `/lib`, `/lib32`, `/lib64`, `/bin`, `/sbin`, `/usr`, `/boot`, `/root`, `/tmp`, `/etc`, `/app`, `/run`, `/proc`, `/sys`, `/dev`, `/var`
- Exceptions from the blacklist: `/run/media`
- These directories are mounted under `/var/run/host`: `/etc`, `/usr`

The reason many of the directories are blacklisted is because they already exist in the sandbox such as `/usr` or are not usable in the sandbox.

Device access

While not ideal, `--device=all` can be used to access devices like controllers or webcams.

dconf access

As of xdg-desktop-portal 1.1.0 and glib 2.60.5 (in the runtime) you do not need direct DConf access in most cases.

As of now this glib version is included in `org.freedesktop.Platform//19.08` and `org.gnome.Platform//3.34`.

If an application existed prior to these runtimes you can tell Flatpak (>= 1.3.4) to migrate the DConf settings on the host into the sandbox by adding `--metadata=X-DConf=migrate-path=/org/example/foo/` to

`finish-args`. The path must be similar to your app-id or it will not be allowed (case is ignored and _ and - are treated equal).

If you are targeting older runtimes or require direct DConf access for other reasons you can use these permissions:

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

With those permissions glib will continue using dconf directly.

1.3.7 Guides

Flatpak provides a range of options and helper tools, which allow building applications using the most common languages and development platforms. These pages provide information on these, and are intended to supplement the standard guidance provided elsewhere in the Flatpak documentation.

Python

Python applications that use supported build systems like Meson, CMake, or Autotools can be built using the standard method. However, many Python applications use custom install scripts or are expected to be installed through Setuptools and `pip`.

For these cases, `flatpak-builder` provides the `simple` buildsystem. Rather than automating the build process, `simple` accepts a `build-commands` array of strings, which are executed in sequence.

For example, the following JSON makes building the popular requests module rather straightforward:

```
{
  "name": "requests",
  "buildsystem": "simple",
  "build-commands": [
    "pip3 install --prefix=/app --no-deps ."
  ],
  "sources": [
    {
      "type": "archive",
      "url": "https://files.pythonhosted.org/packages/source/r/requests/requests-2.18.
↪4.tar.gz",
      "sha256": "9c443e7324ba5b85070c4a818ade28bfabedf16ea10206da1132edaa6dda237e"
    }
  ]
}
```

Here, `build-commands` is an array containing the commands required to build and install the module. As can be seen, in this case `pip` is run to do this. Here, the `--prefix=/app` option is important, because otherwise `pip` would try to install the module under `/usr/` which, because `/usr/` is mounted read-only inside the sandbox, would fail.

Note that `--no-deps` is only used for the purpose of the example - since the requests module has its own dependencies, the build would fail. If multiple dependencies are required, it is better to install them using the method in the next section, instead.

Building multiple python dependencies

Even though the example above installs, it won't actually work. This is because the requests module has a number of dependencies that haven't been installed:

- certifi
- chardet
- idna
- urllib3

Four dependencies aren't very many, and could be installed these using the `simple` method described above. However, anything more complex than this would quickly become tedious.

For these cases, flatpak-pip-generator can be used to generate the necessary manifest JSON. This is a Python script that takes a package name and uses `pip` to identify its dependencies, along with their tarball URLs and hashes.

Using `flatpak-pip-generator` is as simple as running:

```
$ python3 flatpak-pip-generator requests
```

This will output a file called `python3-requests.json`, containing the necessary manifest JSON, which can then be included in your application's manifest file.

Electron

Due to the nature of Electron, building Electron applications as Flatpaks requires a few extra steps compared with other applications. Thankfully, several tools and resources are available which make this much easier.

This guide provides information on how building Electron applications differs from other applications. It also includes information on the tooling for building Electron applications and how to use it.

The guide walks through the manifest file of the sample Electron Flatpak application. Before you start, it is a good idea to take a look at this, either online or by downloading the application.

Building the sample application

While it isn't strictly necessary, you might want to try building and running the sample application yourself.

To get setup for the build, download or clone the sample app from GitHub, and navigate to the `/flatpak` directory in the terminal. You must also install the Electron base app:

```
$ flatpak install flathub io.atom.electron.BaseApp//stable
```

Then you can run the build:

```
$ flatpak-builder build org.flathub.electron-sample-app.json --install
```

Finally, the application can be run with:

```
$ flatpak run org.flathub.electron-sample-app
```

Basic configuration

The first part of the sample application's manifest specifies the application's ID. It also configures the runtime and SDK:

```
"app-id": "org.flathub.electron-sample-app",
"runtime": "org.freedesktop.Platform",
"runtime-version": "1.6",
"branch": "stable",
"sdk": "org.freedesktop.Sdk",
```

The Freedesktop runtime is generally the best runtime to use with Electron applications, since it is the most minimal runtime, and other dependencies will be specific to Electron itself.

The Electron base app

Next, the manifest specifies that the Electron base app should be used, by specifying the `base` and `base-version` properties in the application manifest:

```
"base": "io.atom.electron.BaseApp",
"base-version": "stable",
```

Base apps are described in building-basics. Using the Electron base app is much faster and more convenient than manually building Electron and its dependencies. It also has the advantage of reducing the amount of duplication on users' machines, since it means that Electron is only saved once on disk.

Note that this base app is for projects using Electron 1.x.x, the most common version at the time of writing. Electron 2.x.x applications should use `org.electronjs.Electron2.BaseApp` instead.

Command

The `command` property indicates that a script called `run.sh` is to be executed to run the application. This will be explained in further detail later.

```
"command": "run.sh",
```

Sandbox permissions

The standard guidelines on sandbox permissions apply to Electron applications. However, Electron does not currently support Wayland, so for display access, only X11 should be used. The sample app also configures pulseaudio for sound and enables network access:

```
"finish-args": [
    "--share=ipc",
    "--socket=x11",
    "--socket=pulseaudio",
    "--share=network"
],
```

Build options

These build options aren't strictly necessary, but can be useful if something goes wrong. `env` allows setting an array of environment variables, in this case we set `NPM_CONFIG_LOGLEVEL` to `info` so that `npm` gives us more detailed error messages.

```
"build-options" : {
    "cflags": "-O2 -g",
    "cxxflags": "-O2 -g",
    "env": {
        "NPM_CONFIG_LOGLEVEL": "info"
    }
},
```

Building Node.js

The next part of the manifest is the modules list. The Electron base app does not include Node.js, so it is necessary to build Node.js as a module. This tutorial builds Node.js 8.11.1, as this version works with most projects at the time of writing, but make sure to use whichever version is best for your project.

```
{
    "name": "nodejs",
    "cleanup": [
        "/include",
        "/share",
        "/app/lib/node_modules/npm/changelogs",
        "/app/lib/node_modules/npm/doc",
        "/app/lib/node_modules/npm/html",
        "/app/lib/node_modules/npm/man",
        "/app/lib/node_modules/npm/scripts"
    ],
    "sources": [
        {
            "type": "archive",
            "url": "https://nodejs.org/dist/v8.11.1/node-v8.11.1.tar.xz",
            "sha256":
↪"40a6eb51ea37fafcf0cfb58786b15b99152bec672cccf861c14d1cca0ad4758a"
        }
    ]
}
```

Here, the cleanup step isn't strictly necessary. However, removing documentation helps to reduce final disk size of the bundle.

The application module

The final section of the manifest defines how the application module should be built. This is where some of the additional logic for Electron and Node.js can be found.

```
"name": "electron-sample-app",
"build-options" : {
    "env": {
        "electron_config_cache": "/run/build/electron-sample-app/npm-cache"
```

(continues on next page)

(continued from previous page)

```
    }
},
```

By default, `flatpak-builder` doesn't allow build tools to access the network. This means that tools which rely on downloading sources will not work. Therefore, Node.js packages must be downloaded prior to running the build. Setting the `electron_config_cache` environment variable means that these will be found when it comes to the build.

The next part of the manifest describes how the application should be built. The simple buildsystem option is used, which allows a sequence of commands to be specified, which are used for the build. The download location and hash of the application are also specified.

```
"buildsystem": "simple",
"sources": [
  {
      "type": "archive",
      "url": "https://github.com/flathub/electron-sample-app/archive/1.0.1.tar.gz",
      "sha256": "a2feb3f1cf002a2e4e8900f718cc5c54db4ad174e48bfcfbddcd588c7b716d5b",
      "dest": "main"
  },
```

Bundling NPM packages

The next line is how NPM modules get bundled as part of Flatpaks:

```
"generated-sources.json",
```

Since even simple Node.js applications depend on dozens of packages, it would be impractical to specify all of them as part of a manifest file. A Python script has therefore been developed to download Node.js packages with NPM and include them in an application's sources.

The Python NPM script requires a `package-lock.json` file. This contains information about the packages that an application depends on, and can be generated by running `npm install --package-lock-only` from an application's root directory (the sample example contains a `package-lock.json`, for reference). The script is then run as follows:

```
$ python3 flatpak-npm-generator.py package-lock.json
```

This generates the manifest JSON needed to build the NPM packages for the application, which are outputted to a file called `generated-sources.json`. The content of this file can be copied to the application's manifest but, because it is often very long, it is often best to link to it from the main manifest, which is done by adding `generated-source.json` as a line in the manifest section, as seen above.

Launching the app

The Electron app is run through a simple script. This can be given any name but must be specified in the manifest's `"command":` property.

```
/* Wrapper to launch the app */
{
  "type": "script",
  "dest-filename": "run.sh",
```

(continues on next page)

(continued from previous page)

```
  "commands": [ "npm start --prefix=/app/main" ]
}
```

Build commands

Last but not least, since the simple build option is being used, a list of build commands must be provided. As can be seen, `npm` is run with the `--offline` option, using packages that have already been cached. These are copied to `/app/main/`. Finally the `run.sh` script is installed to `/app/bin/` so that it will be on `$PATH`:

```
"build-commands": [
    /* Install npm dependencies */
    "npm install --prefix=main --offline --cache=/run/build/electron-sample-app/npm-
↪cache/",
    /* Bundle app and dependencies */
    "mkdir -p /app/main /app/bin",
    "cp -ra main/* /app/main/",
    /* Install app wrapper */
    "install run.sh /app/bin/"
]
```

1.4 Debugging

This section includes documentation on how to debug Flatpak apps.

1.4.1 Running debugging tools

Because Flatpak runs each application inside a sandbox, debugging tools can't be used in the usual way, and must instead be run from inside the sandbox. To get a shell inside an application's sandbox, it can be run with the `--command` option:

```
$ flatpak run --command=sh --devel <application-id>
```

This creates a sandbox for the application with the given ID and, instead of running the application, runs a shell inside the sandbox. From the shell prompt, it is then possible to run the application. This can also be done using any debugging tools that you want to use. For example, to run the application with `gdb`:

```
$ gdb /app/bin/<application-binary>
```

This works because the `--devel` option tells Flatpak to use the SDK as the runtime, which includes debugging tools like `gdb`. The `--devel` option also adjusts the sandbox setup to enable debugging.

Note: The Freedesktop SDK (on which many others are based), includes a range of debugging tools, such as `gdb`, `strace`, `nm`, `dbus-send`, `dconf`, and many others.

`gdb` is much more useful when it has access to debug information for the application and the runtime it is using. Flatpak splits this information off into debug extensions, which you should install before debugging an application:

```
$ flatpak install <runtime-id>.Debug
```

When the `--devel` option is used, Flatpak will automatically use any matching debug extensions that it finds.

It is also possible to get a shell inside an application sandbox without having to install it. This is done using `flatpak-builder`'s `--run` option:

```
$ flatpak-builder --run <build-dir> <manifest> sh
```

This sets up a sandbox that is populated with the build results found in the build directory, and runs a shell inside it.

1.4.2 Creating a .Debug extension

Like many other packaging systems, Flatpak separates bulky debug information from regular content and ships it separately, in what is called a `.Debug` extension.

When an application is built, `flatpak-builder` automatically creates a `.Debug` extension. This can be disabled with the `no-debuginfo` option.

1.4.3 Overriding sandbox permissions

It is sometimes useful to have extra permissions in a sandbox when debugging. This can be achieved using the various sandbox options that are accepted by the run command. For example:

```
$ flatpak run --devel --command=sh --system-talk-name=org.freedesktop.login1
↪<application-id>
```

This command runs a shell in the sandbox for the given application, granting it system bus access to the bus name owned by logind.

1.4.4 Inspecting portal permissions

Flatpak has a number of commands that allow to manage portal permissions for applications.

To see all portal permissions of an application, use:

```
$ flatpak permission-show <application-id>
```

To reset all portal permissions of an application, use:

```
$ flatpak permission-reset <application-id>
```

1.4.5 Interacting with running sandboxes

You can see all the apps that are currently running in Flatpak sandboxes (since 1.2):

```
$ flatpak ps
```

And, if you need to, you can terminate one by force (since 1.2):

```
$ flatpak kill <application-id>
```

1.5 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.

1.5.1 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 to 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.

1.5.2 Hosting a repository

The section on *Flatpak Builder* describes how to generate repositories. The resulting repository can be hosted on a web server for consumption by users.

Important details

Flatpak 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.

Flatpak 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 `flatpak 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
```

1.5.3 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
```

You can also set a runtime repo in the bundle:

```
$ flatpak build-bundle ~/repositories/apps dictionary.flatpak org.gnome.Dictionary --
↪runtime-repo=https://flathub.org/repo/flathub.flatpakrepo
```

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

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

Alternatively, bundles can also be installed directly without importing them:

```
$ flatpak install dictionary.flatpak
```

1.6 Desktop Integration

Requirements & Conventions covers the essential aspects of Linux desktop integration. This page provides further information on optional desktop integration features. It also provides guidance on how applications can ensure that their user interfaces fit into the whole range of Linux desktops and distributions.

This information is primarily intended for developers who are new to Linux. However it is also relevant to desktop-specific Linux applications who wish to target a broader range of Linux environments.

While targeting the Linux desktop ecosystem might seem challenging, the existence of common standards, in combination with these guidelines, means that supporting the full range of Linux environments needn't be difficult.

1.6.1 Locale detection

Application toolkits, such as Electron, GTK and Qt, provide built-in support for detecting which locale to use. Otherwise, the `setlocale` function can be used.

1.6.2 Portals

Portals are the framework for securely accessing resources from outside an application sandbox. They provide a range of common features to applications, including:

- Determining network status
- Opening a file with a file chooser
- Opening URIs
- Preventing the device from suspend/sleep/powering off
- Printing
- Sending email
- Showing notifications
- Taking screenshots and screencasts

Toolkits like GTK and Qt provide transparent support for portals. See portals-gtk or portals-qt for detailed information about GTK and portals. If you are not using one of these toolkits, it is possible to access the portals API directly. See the portals API documentation for more information.

1.6.3 Notifications

A number of toolkits and frameworks provide transparent support for Linux desktop notifications. This includes Electron, GTK, KDE and QML.

1.6.4 Status icons

Status icons are the same concept as the system tray or the taskbar on Windows, or menu bar icons on Mac. These are supported on most Linux distributions, through libappindicator.

A number of Linux distributions don't show status icons. It is still possible to provide a status icon, and it will be shown in some distributions. However, in order to ensure compatibility, it is recommended to only use status icons in a supplementary manner, and not to rely on them as the only mechanism for providing status information or access to particular features. This includes "minimize to tray" (or equivalent) functionality.

XEmbed style icons will function with the `x11` permission but all other status icon interfaces require extra permissions to escape the sandbox and these services are not designed to be robust against untrusted software.

1.6.5 System search

GNOME-based distributions, like CentOS, Fedora, Red Hat Enterprise Linux and Ubuntu, provide the option to integrate with system search, by providing a search provider. This allows application-provided search results to appear in the Activities Overview.

1.6.6 Window controls

Window controls are the buttons used to close, maximize and minimize windows. These do vary across Linux desktops, particularly in terms of which controls are shown. Whether applications attempt to follow these variations is up to their discretion. Providing the exact same controls as used by a particular desktop environment should not be seen as a hard requirement.

From a user experience perspective, it is important to ensure that window controls appear on the same side of the window as other desktops. On Linux this is the right side of the window (like Windows).

Applications can rely on system-provided titlebars on Linux, if they don't want to draw their own window controls.

1.6.7 Window decorations

If your application uses a dark visual style as well as system-provided window decorations, the `GTK_THEME_VARIANT=dark` X11 window property should be used, to ensure that window decorations match the rest of the application window. This can be done by running:

```
xprop -f _GTK_THEME_VARIANT 8u -set _GTK_THEME_VARIANT dark
```

1.6.8 Global menus

If your application uses the built in `GtkApplication:menu-bar` or the Qt 5 equivalent they will work as expected from within a sandboxed application.

1.7 Tips and Tricks

This page explains a few useful features of the Flatpak CLI.

1.7.1 Testing an app with a different runtime

You can (for testing) run an application with a different runtime than it typically uses. For instance, to run stable gedit with the latest unstable gnome runtime you can do:

```
$ flatpak run --runtime-version=master org.gnome.gedit
```

You can also use a completely different runtime (but same version number):

```
$ flatpak run --runtime=org.gnome.Sdk org.gnome.gedit
```

If you just want to use the sdk instead of the platform like the above, a better approach is to use `-d`.

Warning: Running against a runtime with a completely different ABI is undefined and unsupported behavior.

1.7.2 Downgrading

It is possible to downgrade an installed application (or runtime) to an older build.

First you look for the commit you are interested in:

```
$ flatpak remote-info --log flathub org.gnome.Recipes
```

Then you deploy the commit:

```
$ sudo flatpak update \
  --commit=ec07ad6c54e803d1428e5580426a41315e50a14376af033458e7a65bfb2b64f0 \
  org.gnome.Recipes
```

If you have Flatpak 1.5.0 or later, you can also prevent the app from being included in updates (either manual or automatic):

```
$ flatpak mask org.gnome.Recipes
```

1.7.3 Bisecting regressions in application builds

In case the newest builds of an application introduce regressions, you can use `flatpak-bisect` to discover which commit introduced the regression. It works just like `git bisect`.

In case your distribution doesn't install the `flatpak-bisect` utility, you can find it distributed alongside the Flatpak source code, in https://github.com/flatpak/flatpak/blob/master/scripts/flatpak-bisect

First you update the application and get its history:

```
$ flatpak-bisect org.gnome.gedit start
```

Then, you should set the current commit as the first bad commit:

```
$ flatpak-bisect org.gnome.gedit bad
```

Now you need to find the hash of the first known good commit. For that, you can see the build history by running:

```
$ flatpak-bisect org.gnome.gedit log
```

To start bisecting, checkout the first known good commit you find:

```
$ flatpak-bisect org.gnome.gedit checkout␣
↪5cd2b0648618c9038fbc6830733817309ade29541cdd8383830bbb76f6accf0d
```

After setting the bad commit and the first known good commit, you can launch the application to verify if the current commit in the bisecting session is a good or a bad one.

To mark a commit as good or bad, run:

```
$ flatpak-bisect org.gnome.gedit good
```

Or:

```
$ flatpak-bisect org.gnome.gedit bad
```

`flatpak-bisect` will inform you when the first bad commit is found.

1.8 Reference Documentation

Reference documentation for flatpak, flatpak-builder and libflatpak.

1.8.1 Flatpak Command Reference

1.8.2 Flatpak Builder Command Reference

1.8.3 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.

What is mentioned here is just a high level look at the contents. To have up to date information simply install the runtime and open a shell inside of it (`flatpak run org.freedesktop.Sdk//18.08`) from there you can look around or use tools like `pkg-config --list-all`. In the runtime shell you can also inspect `/usr/manifest.json`, which lists the sources used to build it.

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, Gtk3, 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)

1.8.4 Sandbox Permissions

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

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

`--socket=x11`	Show windows using X11
`--share=ipc`	Share IPC namespace with the host[1]
`--allow=bluetooth`	Allow access to Bluetooth
`--device=dri`	OpenGL rendering
`--socket=wayland`	Show windows using Wayland
`--socket=pulseaudio`	Play sounds using PulseAudio
`--share=network`	Access the network **[#f2]_**
`--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

Filesystem permissions

Each of the following permissions configure filesystem access, and should be added to `--filesystem=`:

[1] This is not necessarily required, but without it the X11 shared

`host`	Access all files
`home`	Access the home directory
`/some/dir`	Access an arbitrary path
`~/some/dir`	Access an arbitrary path relative to the home directory
`xdg-desktop`	Access the XDG desktop directory
`xdg-documents`	Access the XDG documents directory
`xdg-download`	Access the XDG download directory
`xdg-music`	Access the XDG music directory
`xdg-pictures`	Access the XDG pictures directory
`xdg-public-share`	Access the XDG public directory
`xdg-videos`	Access the XDG videos directory
`xdg-templates`	Access the XDG templates directory
`xdg-config`	Access the XDG config directory
`xdg-cache`	Access the XDG cache directory
`xdg-data`	Access the XDG data directory
`xdg-run/path`	Access subdirectories of the XDG runtime directory (where path is any subdirectory)

Paths can be added to all the above filesystem options. For example, `--filesystem=xdg-documents/path`. The following permission options can also be added:

- `:ro` - read-only access
- `:rw` - read/write access (this is the default)
- `:create` - read/write access, and create the directory if it doesn't exist

memory extension will not work, which is very bad for X11 performance. .. [#f2] 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.

1.8.5 Freedesktop quick reference

In order to ensure interoperability, flatpak adheres strictly to a number of freedesktop standards and practices. This page describes the basic conventions that should be followed when building a flatpak app.

Icons

Application icons can be in either png or svg format, must use the application's appid as a prefix and be placed in `/app/share/icons/hicolor/$size/apps/`

Example:

```
/app/share/icons/hicolor/128x128/apps/org.gnome.Dictionary.png
```

If interested, you can read the full spec here

Desktop files

Desktop files are used by desktop environments in order to identify and display available applications to the user, they contain information about how to launch the application, its icon and categories among others.

A minimal desktop file needs at least the application's *Name*, *Exec* command, *Type* and *Icon*:

```
[Desktop Entry]
Name=Gnome Dictionary
Exec=org.gnome.Dictionary
Type=Application
Icon=org.gnome.Dictionary
```

Your desktop file should be prefixed with your application's appid and placed in `/app/share/applications/`, you should use `desktop-file-validate` to check your file for errors before including it.

Example:

```
/app/share/applications/org.gnome.Dictionary.desktop
```

You can find more information here. If interested, you can read also the full spec here.

Appdata files

Appdata files are used by application stores (eg. kde discover, gnome software) in order to display metadata about your application, such as a description, screenshots, display changelogs on update, among other things.

your desktop file should be prefixed with your application's appid and placed in `/app/share/metainfo/`, you should also use `appstream-util validate-relax` to check your file for errors before including it.

Example:

```
/app/share/metainfo/org.gnome.Dictionary.appdata.xml
```

If interested, you can read the full spec here

1.8.6 Under the Hood

This page provides an overview of how Flatpak works internally. While it isn't necessary to be familiar with this in order to use Flatpak, some people might find it interesting. Knowing about Flatpak's architecture also helps to get a better understanding of how and why it works the way it does, from a user and application developer perspective.

"Git for apps"

Flatpak is built on top of a technology called OSTree, which is influenced by and very similar to the Git version control system. Like Git, OSTree allows versioned data to be tracked and to be distributed between different repositories. However, where Git is designed to track source files, OSTree is designed to track binary files and other large data.

Internally, Flatpak therefore works in a similar way to Git, and many Flatpak concepts are analogous to Git concepts. Like Git, Flatpak uses repositories to store data, and it tracks the differences between versions.

With Flatpak, each application, runtime and extension is a branch in a repository. An identifier triple, such as `com.company.App/i386/stable` is a reference to that branch. The output of a Flatpak build process is a directory of files which is committed to one of these branches.

When an application is installed with Flatpak, it is pulled from the remote into a new branch in a local repository. Links are then generated which point from the right places in the filesystem to the application's files in the repository (these are hard links, which are fast to resolve and disk space efficient). In other words, every application that is installed is stored in a local version control repository, and is then mapped into the local filesystem.

Version tracking is therefore a core part of Flatpak's architecture, and this makes updating software to the latest version very efficient. Versioning also makes rollbacks possible, so it's easy to go back to a previous version, should that be required.

Storing applications in a local OSTree repository has other advantages. For example, it allows files that are stored on disk to be deduplicated, so the same file that belongs to multiple applications (or runtimes) is only stored once.

Underlying technologies

Flatpak utilises a number of pre-existing technologies. These 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

1.8.7 libflatpak API Reference

1.8.8 Portal API Reference