flatpak test Documentation

Release latest

January 16, 2017

Contents

This covers everything you need to know to build and distribute applications as Flatpaks. The docs start with introduction to the basic concepts and a simple app building tutorial, before moving on to cover automated building and repository hosting.

Example tutorials are used throughout. To complete them, it is necessary to have `flatpak` and `flatpak-builder` installed on your system. flatpak.org provides **'details on how to do this< http://flatpak.org/getting.html>'_**.

CHAPTER 1

Contents

1.1 Introduction to Flatpak

1.1.1 Key concepts

Flatpak is best understood through its key concepts: runtimes, bundled libraries, SDKs and sandboxes. These help to explain how Flatpak differs from traditional application distribution on Linux, as well as the framework's capabilities.

Runtimes

Runtimes provide the environment that each application runs in, including the basic dependencies they might require. Each runtime can be thought of as a `/usr` filesystem (indeed, when an app is run, its runtime is mounted at `/usr`). Various runtimes are available, from more minimal (but more stable) Freedesktop runtimes, to larger runtimes produced by desktops like GNOME or KDE. (The runtimes page provides an overview of the runtimes that are currently available.)

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

Flatpak identifies runtimes (as well as SDKs and applications) by a triple of name/arch/branch. The name is expected to be in inverse-dns notation, which needs to match the D-Bus name used for the application. For example: `org.gnome.Sdk/x86_64/3.14` or `org.gnome.Builder/i386/master`.

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 apps 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.

Both runtimes and app bundles can be installed per-user and system-wide.

SDKs (Software Developer Kits)

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

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.

1.1.2 Technologies

Flatpak tries to avoid reinventing the wheel. We build on existing technologies where it makes sense. Many of the important ingredients for Flatpak are inherited from Linux containers and related initiatives:

- 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 our sandbox
- D-Bus, a well-established way to provide high-level APIs to application
- 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 that makes Flatpak apps show up nicely in software-center applications

1.1.3 The flatpak command

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

Most flatpak commands are performed system-wide by default. To perform a command for the current user only, use the `--user` option.

1.2 Anatomy of a Flatpak Application

Each Flatpak app has the following basic structure:

- `metadata` - a keyfile which provides information about the application, including information that is necessary for setting up the sandbox for running the application
- `/files` - the files that make up the application
- `/files/bin` - application binaries
- `/export` - files which the host environment needs access to, such as the application's desktop file, icon and D-Bus service file

A typical metadata file looks like this:

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

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

[Environment]
GEDIT_FOO=bar

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

This specifies the name of the application, the runtime it requires, the SDK that it is built against and the command used to run it. It also specifies file and device access, sets certain environment variables (inside the sandbox, of course), and how to connect to the session bus.

All the files in the export directory must have the application id as a prefix. This guarantees that applications cannot cause conflicts, and that they can't override any system installed applications.

1.2.1 AppData

Many Linux distributions provide an app store or app center for browsing and installing applications. AppData is a standard format for providing application information that can be used by app stores, such as an application description and screenshots. Flatpak makes use of the AppData standard, and application authors are recommended to use it to include information about their applications.

1.2.2 Extensions

Applications and runtimes can define extension points, where optional pieces can be plugged into the filesystem. Flatpak is using this to separate translations and debuginfo from the main application, and to include certain parts that are provided separately, such as GL libraries or gstreamer plugins.

When flatpak is setting up a sandbox, it is looking for extension points that are declared in the application and runtime metadata, and mounts runtimes with a matching name. A typical extension section in a metadata file looks like this:

```
[Extension org.gnome.Platform.GL]
version=1.4
directory=lib/GL
```

More complicated extension points can accept multiple extensions that get mounted below a single directory. For example, the gstreamer extension:

```
[Extension org.freedesktop.Platform.GStreamer]
version=1.4
directory=lib/extensions/gstreamer-1.0
subdirectories=true
```

The `subdirectories=true` key instructs flatpak to mount e.g. a `org.freedesktop.Platform.GStreamer.mp3` runtime on `/usr/lib/extensions/gstreamer-1.0/mp3`

in the sandbox. The gstreamer libraries in the `org.freedesktop.Platform` runtime have been configured to look in this place for plugins.

1.3 Building Simple Apps

The `flatpak` utility provides a simple set of commands for building and distributing applications. These allow creating new Flatpaks, into which new or existing applications can be built. This section describes how to build a simple application which doesn't require any additional dependencies outside of the runtime it is built against.

1.3.1 Installing an SDK

As described above, an SDK is a special type of runtime that is used to build applcations. Typically, an SDK is paired with a runtime that will be used by the app at runtime. For example the GNOME 3.22 SDK is used to build applications that use the GNOME 3.22 runtime. The rest of this guide uses this SDK and runtime for its examples. To do this, download the repository GPG key and then add the repository that contains the runtime and SDK:

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

You can now download and install the runtime and SDK. (If you have already completed the tutorial on the Flatpak homepage, you will already have the runtime installed.)

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

This might be a good time to try installing an application and having a look 'under the hood'. To do this, you need to add a repository that contains applications. In this case we are going to use the gnome-apps repository and install gedit:

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

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

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

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

1.3.2 Creating an app

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

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

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

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

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

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

1.3.3 Building

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

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

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

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

```
$ wget https://download.gnome.org/sources/gnome-dictionary/3.22/gnome-dictionary-3.22.0.tar.xz
$ tar xvf gnome-dictionary-3.22.0.tar.xz
$ cd gnome-dictionary-3.22.0/
```

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

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

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

1.3.4 Completing the build

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

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

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

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

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

1.4 Flatpak Builder

If an application requires additional dependencies that aren't provided by its runtime, Flatpak allows them to be bundled as part of the app itself. This requires building each module inside the application build directory, which can be a lot of work. The `flatpak-builder` tool can automate this multi-step build process.

flatpak-builder takes care of the routine commands used to build an app and any bundled libraries, thus allowing application building to be automated. To do this, it expects modules to be built in a standard manner by following what is called the Build API. If any modules don't conform to this API, they will need to be modified.

1.4.1 Manifests

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

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

```
{
  "app-id": "org.gnome.Dictionary",
  "runtime": "org.gnome.Platform",
  "runtime-version": "3.22",
  "sdk": "org.gnome.Sdk",
  "command": "gnome-dictionary",
  "finish-args": [
     "--socket=x11",
     "--share=network"
  ],
  "modules": [
    {
      "name": "gnome-dictionary",
      "sources": [
        {
          "type": "archive",
          "url": "https://download.gnome.org/sources/gnome-dictionary/3.22/gnome-dictionary-3.22.0.ta
          "sha256": "efb36377d46eff9291d3b8fec37baab2355f9dc8bc7edb791b6a625574716121"
        }
      ]
    }
  ]
}
```

1.4.2 Cleanup

flatpak-builder performs a cleanup phase after the build, which can be used to remove headers and development docs, among other things. Two properties in the manifest file can be 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, and will then only match filenames that were created by that particular module.

1.4.3 File renaming

Files that are exported by a flatpak must be named using the application ID. However, application's source files will typically not follow this convention. To get around this, flatpak-builder allows renaming application icons, desktop files and AppData files as a part of the build process, using the `rename-icon`, `rename-desktop-file` and `rename-appdata` properties.

1.4.4 Splitting things up

By default, flatpak-builder splits off translations into a separate .Locale runtime, and debuginfo into a .Debug runtime, and adds these 'standard' extension points to the application metadata. You can turn this off with the `separate-locales` and `no-debuginfo` keys, but there shouldn't be any reason for it.

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

1.4.5 Example

You can try flatpak-builder for yourself, using the repository that was created in the previous section. To do this, place the manifest json from above into a file called `org.gnome.Dictionary.json` and run the following command:

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

This will:

- Create a new directory (called dictionary2)
- Download and verify the Dictionary source code
- Build and install the source code, using the SDK rather than the host system
- Finish the build, by setting permissions (in this case giving access to X and the network)
- Export the resulting build to the tutorial repository, which contains the Dictionary app that was previously installed

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

If you completed the tutorial in **'Building Simple Apps'_**, it is now possible to update the installed version of the Dictionary application with the new version that was built and exported by flatpak-builder:

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

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

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

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

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

1.4.6 Example manifests

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

1.5 Working with the Sandbox

By default, a flatpak has extremely limited access to the host environment. This includes:

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

Most applications will need access to some of these resources in order to be useful, and flatpak provides a number of ways to give an application access to them. The `build-finish` command is the simplest of these. As was seen in a previous example, this can be used to add access to graphics sockets and network resources:

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

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

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

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

Note that in this example access to the filesystem wasn't granted. This can be tested by installing the resulting application and running:

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

build-finish allows a whole range of resources to be added to an application. Run `flatpak build-finish --help` to view the full list.

There are several ways to override the permissions that are set in an application's metadata file. One of these is to override them using flatpak run, which accepts the same parameters as `build-finish`. For example, this will let the Dictionary application see your home directory:

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

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

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

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

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

1.5.1 Useful sandbox permissions

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

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

1.6 Hosting a repository

While it is relatively simple to host a flatpak repository, there are some important details to be aware of.

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

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

1.6.1 GPG signatures

By default OSTree refuses to pull from a remote repository that is not signed. To disable GPG verification, the `--no-gpg-verify` option needs to be used when a remote is added. Alternatively, it can be disabled on an existing remote using `flatpak remote-modify`.

Note that GPG signatures are required for the user to be able to install trusted remotes that can be updated from without needing to be root.

OSTree requires signatures for every commit and on repository summary files. These objects are created by the `build-update-repo` and `build-export` commands, as well as indirectly by `flatpak-builder`. A GPG

[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.

key should therefore be passed to each of these commands, and optionally the GPG home directory to use. For example:

```
$ flatpak build-export --gpg-sign=KEYID --gpg-homedir=/some/dir appdir repo
```

1.6.2 Referring to repositories

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

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

A typical `.flatpakrepo` file looks like this:

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

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

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

A typical `.flatpakref` file looks like this:

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

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

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

1.6.3 Single-file bundles

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

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

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

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

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

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

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